名家经典系列

华章科技

HZ BOOKS

PEARSON

由在IBM工作50余年的资深计算机专家撰写，Amazon全五星评价，算法领域最有影响力的著作之一

Google公司首席架构师、Jolt大奖得主Joshua Bloch和Emacs合作创始人、C语言畅销书作者Guy Steele倾情推荐

算法的艺术和数学的智慧在本书中得到了完美体现，书中总结了大量高效、优雅和奇妙的算法，并从数学角度剖析了其背后的原理

算法心得

高效算法的奥秘

（原书第2版）

（美）Henry S. Warren，Jr. 著

爱飞翔 译

Hacker's Delight

(Second Edition)

机械工业出版社
China Machine Press

名家经典系列

算法心得

高效算法的奥秘

（原书第2版）

（美）Henry S. Warren，Jr. 著
爱飞翔 译

Hacker's Delight

(Second Edition)

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

算法心得：高效算法的奥秘（原书第2版）/（美）沃伦（Warren，Jr.，H. S.）著；爱飞翔译. —北京：机械工业出版社，2014.1

（名家经典系列）

书名原文：Hacker's Delight，Second Edition

ISBN 978-7-111-45356-7

Ⅰ. 算… Ⅱ. ①沃… ②爱… Ⅲ. 电子计算机一算法理论 Ⅳ. TP301.6

中国版本图书馆CIP数据核字（2014）第002804号

本书版权登记号：图字：01-2013-0212

本书是算法领域最有影响力的著作之一，与大师高德纳所著的《计算机程序设计艺术》共同被誉为所有程序员都应该阅读的计算机著作。它由在IBM工作50余年的资深计算机专家撰写，Amazon全五星评价，Google公司首席架构师、Jolt大奖得主Hoshua Bloch和Emacs合作创始人、C语言畅销书作者Guy Steele倾情推荐。书中总结了大量高效、优雅和奇妙的算法，并从数学角度剖析了其背后的原理，算法的艺术和数学的智慧在本书中得到了最好的体现。

本书共18章。第1章是概述；第2章介绍了基础知识；第3～4章介绍了2的幂边界和算术边界；第5章讨论了位计数；第6～7章讲解了在字组中搜索位串和重排位元与字节；第8～10章分别介绍了乘法、整数除法和以除数为常量的整数除法；第11章讲解了初等函数；第12章介绍了以特殊值为底的数制；第13～15章分别讲解了格雷码、循环冗余校验和纠错码；第16～18章分别介绍了希尔伯特曲线、浮点数和素数公式。之后是各章习题的参考答案。附录分别介绍了计算机算术运算表、牛顿法和各种离散函数图像。

机械工业出版社（北京市西城区百万庄大街22号 邮政编码100037）

责任编辑：关 敏

北京市荣盛彩色印刷有限公司印刷

2014年3月第1版第1次印刷

186mm×240mm·27.25印张

标准书号：ISBN 978-7-111-45356-7

定 价：89.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991 88361066 投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

译者序

写代码总会遇到难题，时而苦于乘法操作频繁溢出，时而苦于开方算法太过笨拙，于是，程序员之间口耳相传的那些代码秘籍，这些时候就该大显身手了。有些小程序，仅两三行代码即能解决平常数十行代码方能实现的功能；还有些小程序，只用 0x24924925 这般神奇的数字，即能成倍提升运算速度。读者若对此感兴趣，则本书定能令你开怀畅读。

作者从事计算机研发工作数十年，他将期间所得之大量技巧融于书中。本书不但讲授算法技巧，而且还会剖析背后的数学原理，令你在学会某个奇妙算法后，可举一反三，推出很多类似技巧，以运用于不同场合。

在研究这些高效而优雅的算法时，作者还会如数家珍地列出许多变体，并旁征博引地讲述可以解决同一问题的其他思路，铺陈完毕后，更会将各自优劣娓娓道来。实际应用中，经常需要权衡各算法之轻重，嵌入式开发、硬件编程、图形渲染、游戏智能等领域尤其如此，若是平素能像作者这样勤于总结、善于对比，那么在需要用到相关技巧时必能信手拈来，左右逢源。

从培养兴趣、锻炼思维、付诸实践三个角度观之，本书皆为精彩而思辨的智慧书。既可静心品读代码之诗意，又能细致体味数学之美感，何其乐哉！

作者乃业界翘楚，学识渊博而思维开阔，文中部分词句与日常用语及数学、计算机等领域一般用法不甚相同，故译文或加注释或添引号，以强调其特殊含义。

翻译过程中，得到机械工业出版社华章公司诸君勉励，于此深表谢意。

本书主要由爱飞翔翻译，舒亚林、张军、王鹏亦参与部分翻译工作。小弟乐意与各位朋友通过个人网站（www.agilemobidev.com）及电子邮件（eastarstormlee@gmail.com）探讨算法问题。由于时间仓促，水平有限，错误与疏漏在所难免，敬请读者不吝赐教。

爱飞翔

2014 年 2 月

序（第 1 版序）

三十年前的一个暑假，我在麻省理工学院的 Project MAC 实验室[一]找到了首份工作。当时很喜欢操作那里的 DEC PDP-10 计算机[二]。这台计算机有丰富且易于操控的指令集，可以做位测试和位屏蔽，并操作位段[三]与整数，所以和其他计算机相比，拿它写汇编语言代码绝对要更顺手些。尽管已经停产多年，仍有一批狂热的粉丝还在玩 PDP-10，还有一些人拿家用电脑模拟 PDP-10 指令集，这样就可以执行以前的整个 PDP-10 操作系统及其附属软件了。这帮人甚至还在该平台上开发新的程序，比如说现在还有些网站用模拟的 PDP-10 平台来提供网页服务。（拜托，大家别笑，与那些爱开老爷车的人相比，这种怀旧方式也不算什么。）

1972 年夏天，还有一件事让我开心，那就是读到了一本全新的 MIT 研究备忘录：HAKMEM。这部秘籍收录了从电路到数论的各种技术细节[四]，而最令我着迷的内容还是其中各项精妙的编程小技巧。通常每一条编程技巧都会讲一个貌似符合编程规范但是看上去却十分怪异的操作。这些整数或位串操作（bit string，例如计算某个字[五]中值为 1 的

[一] “数学与计算研究计划”（Project on Mathematics and Computation）一词的缩写，是 1963 年设立于麻省理工学院（Massachusetts Institute of Technology，MIT）的一个研究项目，为当前“MIT 计算机科学与人工智能实验室”（MIT Computer Science and Artificial Intelligence Laboratory）的前身。详情参见：https://en.wikipedia.org/wiki/Project_MAC#Project_MAC。——译者注

[二] DEC 公司（Digital Equipment Corporation）于 20 世纪 60 至 80 年代推出的一系列大型计算机。详情参见：https://en.wikipedia.org/wiki/PDP-10。——译者注

[三] field，在这里指 bit field，也叫“位域”，是一种数据结构，可将各个字段用若干个位元的形式紧密存储，以节省空间。详情参见：https://zh.wikipedia.org/wiki/位段。——译者注

[四] “HAKMEM”是“hacks memo”的简称，为什么这样写呢？因为 PDP-10 平台的字（word）长度为 36 位，而每个字符占 6 个二进制位，这样，一个字就能容纳 6 个字符，所以很多 PDP-10 程序员用的名字都是 6 个字符，我们这帮人一看到 6 字符的缩写，立马就能还原出它的原意。所以，“HAKMEM”这个怪名字在那个时代是合理的，至少对程序员来说是这样。

[五] word，在计算机领域里指由一定数量的二进制位（bit，比特）所构成的数据结构。为了与日常汉语中的“字”区别开，译文多以“字组”称之。详情参见：https://zh.wikipedia.org/wiki/字_（计算机）。此外，为了与十进制的“数位”（digit）和常用的“位置”等词相区分，译文将酌情以“位元”一词对应 bit。——译者注

二进制位共有多少个），本来用一个较长但是约定俗成的机器指令序列或是循环就能轻松实现，然而秘籍中却用了一种极为巧妙的办法来完成：它只用三四个甚至两个精心选择的指令就够了。这些指令的意思很隐晦，需要有人给你解释，或是自己仔细研究才能搞清楚。品尝这份编程大餐，与吃花生豆或夹心软糖那种小零食一样，也令人欲罢不能。这些技巧内涵丰富，充满了深邃的智慧，体现了程序的美感，甚至让人读出了诗意。

我心里当时就在想："这种技巧不会只有这么几条吧？"我把这些年来在各种场合发现的编程心得总结了一下，果然数量非常多。所以我又想，"是不是该有一本专著来谈谈这个问题呢？"

看到 Hank Warren 先生的原稿时，我实在太高兴了：他将这些编程小技巧系统地收集起来，按主题分类，并清晰地讲解其意义。虽说有些技巧是用机器指令描述的，但是这本书可不是只写给汇编语言程序员看的。本书主题是探讨计算机中整数与位串的基本结构关系，以及如何才能更为高效地操作它们。汇编语言中能用的技巧，放在 C 或 Java 语言里照样适用。

很多算法和数据结构的书都在讲排序和搜索这些复杂的技术，告诉你如何维护哈希表和二叉树，怎样操作记录和指针等，然而它们都没有讲到二进制位及位元数组这种微型数据结构的功用。单用二进制加减法和位操作就能实现很多强大的功能。由于进位链（carry chain）机制，改变一个位元的值有可能会影响到它左方的所有二进制位。某些利用此机制的二进制加法技巧不为人熟知，然而它们却是操作数据的利器。

的确该有一本书来总结这些技巧了。你手中捧着的这本就是，而且写得非常好。要想优化编译器或编写高效代码，那非读它不可。也许这些技巧不是每天都能用到，但在遇到困难时，还是会派上用场的。有时我们要遍历一个字中的位元，有时要执行某些不太好实现的整数操作，有时为了让运行速度加倍，还要优化内循环[⊖]中那些棘手的整数或位元运算——如果真碰到上述情况，那就得求助此书啦。当然了，有时也不必想那么多，直接翻开来读，就必能体会到个中乐趣。

Guy L. Steele，Jr.

2002 年 4 月于马萨诸塞州伯灵顿

⊖ inner loop，指两重循环中内部的那一层，通常优化程序时应首先考虑减少内层循环体的执行时间。详情参见：https：//en. wikipedia. org/wiki/Inner _ loop。——译者注

前　言

程序员创造力第一定律：软件维护成本与程序员创造力的平方成正比。

——Robert D. Bliss，1992 年

本书收集了笔者多年来总结的一些编程小技巧，其中大部分都必须运行在以二进制补码来表示整数的电脑[一]上。尽管本书假设寄存器长度是 32 位，但在寄存器长度不是 32 位的电脑中，这些技巧基本上仍能适用。

本书不打算讲高深的排序算法或是编译器的优化技术等大话题，而是着重来谈涉及单个计算机字组或指令的小技巧，比如怎样统计字组中值为 1 的位元数。此类技巧通常需要混用算术与逻辑指令。

我们还需假定整数溢出[二]中断已经屏蔽，这样的话，就算整数运算溢出了，也不会出问题。C、Fortran、Java 等程序都是如此，但使用 Pascal 与 Ada 语言的程序员一定要注意这一点!

书中以通俗的语言来讲述各技巧，只有算法含义不明显时才会给出证明，有时甚至略去证明。算法中将会出现计算机算术指令、“地板”函数[三]、算术与逻辑操作混搭等内容，它们要证明起来通常比较困难，而且也不易表述。

笔者把很多算法都写成了程序代码，在电脑上执行并验证，以减少笔误及疏忽。用实际存在的编程语言来描述算法，就有这个好处；然而即便如此，每一种计算机语言也还是各有其缺陷。我使用很多人能看懂的 C 代码来描述高级语言部分，因为它可以直接

㊀ 在本书语境中，电脑、计算机（computer）、机器（machine）、处理器（processor）等词常常都指代中央处理器（CPU），故译文在不致混淆的情况下，将其视为互相通用的概念。——译者注

㊁ integer overflow，指算术运算时由于结果过大或过小而超出存储范围的情形。详情参见：https://en.wikipedia.org/wiki/Integer_overflow。——译者注

㊂ floor function，又叫向下取整函数或最低值函数，返回不大于自变量的最大整数，也常称为“高斯函数”。详情参见：https://zh.wikipedia.org/wiki/地板函数。——译者注

把整数与位串操作写在一起，而且很多 C 语言编译器都能产生高质量的目标代码㊀。

书中偶尔也会用到机器语言，它们都以“三地址格式”㊁出现，这样读起来更方便。其中的汇编语言是笔者参照当下的 RISC 指令集㊂虚构出来的。

大家应该尽量编写没有分支的代码（branch-free code），因为分支代码会拖慢很多电脑的指令获取速度，并阻止 CPU 并发执行。此外，它还会妨碍一些编译器优化措施㊃，如指令调度㊄、重复子表达式消除㊅、寄存器分配㊆等。也就是说，编译器优化大段的简单代码要比优化很多小代码块的效果更好。

此外，编码时还应该多用数值较小的常量㊇，多与 0 比较（少与非 0 值比较），尽量写出易于并发执行的指令序列㊈。如果改写书里很多代码，让它们从内存中查表，其结果会更精确，然而笔者通常不提这种做法。因为与算术指令相比，从内存中加载数据反而更耗时。虽说查表法的确很实用，但是一般来说都比较乏味。当然还有些特例不属于上述情况。

最后要说的是，本书书名㊉中的“hacker”用的是本意，也就是指痴迷于计算机的人。这种人喜欢拿电脑开发点新玩意儿，或是用新潮而有创意的办法来实现原有的功能。

㊀ object code，又叫“目的码”，是编译器处理源代码后的产物，一般由机器码或近于机器语言的代码组成。存放此类代码的文件叫目标文件（object code），也叫二进制文件。详情参见：https://zh.wikipedia.org/wiki/目标代码。——译者注

㊁ three-address format，即 three address code，缩写为 TAC 或 3AC，又叫“三位址码”、“三地址码”。它把通常的算式“运算结果=操作数 1 操作符 操作数 2”写成“操作符 操作数 1，操作数 2，运算结果”的形式，因其中牵涉 3 个变量（即操作数 1、操作数 2、运算结果）而得名。例如 $z=x+y$ 用 TAC 格式来写就是 add x，y，z。详情参见：https://en.wikipedia.org/wiki/Three_address_code。——译者注

㊂ RISC，Reduced Instruction Set Computing 的缩写，是一种设计 CPU 的模式，该设计思路精简了指令数目及定址方式，使 CPU 的制作更为容易，也有助于提升其并行能力与执行效率。使用此种指令集的 CPU 目前主要应用于智能手机和平板电脑等移动设备上，与之相对的是 x86 等处理器所用的“复杂指令集”（CISC，C 表示 Complex）。详情参见：https://zh.wikipedia.org/wiki/精简指令集。——译者注

㊃ 常见的编译器优化技法请参考：https://en.wikipedia.org/wiki/Category:Compiler_optimizations。——译者注

㊄ instruction scheduling，是通过指令流水线（instruction pipeline）技术提升指令并发执行效果的优化手法。详情参见：https://en.wikipedia.org/wiki/Instruction_scheduling。——译者注

㊅ commoning，即 common subexpression elimination（CSE），是通过消减重复的表达式来提升执行速度的优化技术。详情参见：https://en.wikipedia.org/wiki/Common_subexpression_elimination。——译者注

㊆ register allocation，该技术让众多变量共用一个 CPU 寄存器，以优化执行速度。详情参见：https://en.wikipedia.org/wiki/Register_allocation。——译者注

㊇ immediate value，也叫“立即值”。——译者注

㊈ instruction-level parallism，缩写为 ILP，意为指令级并行度，用于度量指令序列的并发执行程度。例如某段指令在非并发方式下需要 3 个时间单位来执行，而并发执行只需 2 个时间单位，那么其 ILP 就是 3/2。详情参见：https://en.wikipedia.org/wiki/Instruction-level_parallelism。书中还用该词表示 CPU 并发执行指令的能力。——译者注

㊉ 本书英文书名为 Hacker's Delight。——译者注

他们都挺会用电脑，但却很可能不是一名专职电脑程序员或设计师，这些电脑极客㊀开发出来的东西，有些有用，有些只是玩玩而已。说到此类纯属玩乐的戏作，让我想起很多资深编程票友都写过一段小程序，它可以把自己照原样打印出来㊁。这才是我使用“hacker”一词的初衷，在书里可别想找到教你怎样入侵他人电脑的把戏哦。

致谢

首先要感谢 Bruce Shriver 与 Dennis Allison 两位先生鼓励我写作本书，然后还要对 IBM 诸位同仁致意，参考文献中也提到了其中一些同事的名字。尤其感谢 Martin E. Hopkins 先生，他可谓 IBM 的“活编译器”(Mr. Compiler)。Hopkins 先生在编码工作中一丝不苟，认真优化每一个循环——这种敬业精神深深感染了我。仰赖 Addison-Wesley 各位评审，本书内容有了极大改观。他们的姓名笔者大多不了解，我唯一记住的是大名鼎鼎的 Guy L. Steele，Jr.。在 50 页书评中，Guy 补充了一些我当时没有谈到的新问题，如位元的打乱与复原，“绵羊与山羊分离操作”（分羊法，sheep and goats operation），等等，而且他提出的一些算法也比我原来用的更高明。Guy 是个非常仔细的人，比方说，我曾把十六进制数 AAAAAAAA 的质因子分解式错写为 $2\times3\times17\times257\times65\,537$，而他发现其中的 3 应该是 5。此外，他还提了一些旨在改善写作风格的建议，并且从不回避书里的细节问题。读者若发现文中有疑似他人捉刀之处㊂，那恐怕得归功于 Guy 了。

H. S. Warren，Jr.

2012 年 6 月于纽约州约克镇

本书英文版相关资料请查阅
www. HackersDelight. org

㊀ 本书语境中的“hacker”可理解成“电脑极客”、“程序玩家”、“编程票友”、“编程达人”、“技术发烧友”等意思，以便与容易引发歧义的“骇客”、“黑客”等通俗叫法区分开。本来 hacker 指计算机技术狂热爱好者，而 cracker 指破坏计算机安全的人，但后者的词义逐渐侵蚀了前者，导致 cracker、hacker 及“骇客、黑客、怪客、剑客”等译名全都成了“计算机安全破坏者”，令 hacker 一词丧失了原意。详情参见：https：//zh. wikipedia. org/wiki/骇客。——译者注

㊁ 比方说，下面这段用 C 语言写的程序代码：
```
main () {char * p="main () {char * p=%c%s%c; (void) printf (p, 34, p, 34, 10);}%c"; (void) printf (p, 34, p, 34, 10);}
```

㊂ 原文为 parallel prefix，意为“并行前置式算法”、“平行前缀算法”，是一套分组归并形式的速算流程。详情参见：https：//www. wikipedia. org/wiki/Prefix _ sum。作者在此处以之比喻他与 Guy 四手联弹的佳话。——译者注

目　　录

值得每一位程序员珍藏的经典

推荐阅读

算法导论（原书第2版）

2006、2007 CSDN、《程序员》杂志评选的十大IT好书之一　算法中的经典权威之作

编译原理（原书第2版）

编译领域无可替代的经典著作，被广大计算机专业人士誉为“龙书”

自动机理论、语言和计算导论（原书第3版）

1996年图灵奖得主经典巨著升级版

分布式系统：概念与设计（原书第4版）

本书是衡量所有其他分布式系统教材的标准

数据库系统概念（原书第5版）

数据库系统方面的经典教材，被美誉为“帆船书”

软件工程：实践者研究方法（原书第6版）

全球上百所大学和学院采用　最受欢迎的软件工程指南

程序员提升个人影响力和软实力必读经典

第 1 章

概述

1.1 记法

本书既不是普通的数学算式教程，也不是单纯的电脑程序算法手册，它讲的是“计算机算术”（computer arithmetic），参与运算的数是长度固定的位串与位元数组[1]。计算机算术中的表达式与普通的数学表达式近似，不同之处在于其中的变量指代的是 CPU 寄存器中的内容，而且计算机算术表达式的值是一串不具备特定符号性的位元。在此类表达式中，操作符可能会以不同方式解读其操作数，例如“比较操作符”（comparison operator）有时会将其操作数当成有正负号的二进制整数，有时却把它视为无符号的二进制整数。为免混淆，本书所列计算机算式以不同的符号来区分上述两种情况。

计算机算式与数学算式的主要差别在于：无论是加减法还是乘法，计算机算式的结果总是要跟 2 的 n 次方取模，n 是指当前机器的字长[2]。此外，计算机算式的运算种类也远多于数学算式：除了基本的四则运算外，还有逻辑与、异或、比较、左移，等等。

如未特别指明，则默认字长为 32 位，且带符号的整数均以“2 补码”[3]形式表示。

---

[1] bit vector，即 bit array，也写作 bitmap、bitset、bit string 等，是由若干位元排列而成的数组，又叫“位向量”、“位矢量”等，详情参见：https://en.wikipedia.org/wiki/Bit_array。译文在不致混淆时，均以“位元数组”称之。——译者注

[2] word size，就是字组中所含的位元个数，也称 word length、word width。在不致混淆的情况下，“字长”、“字宽”均指这一概念。——译者注

[3] two's complement，也叫“2 的补码”、“二补数”，是一种用二进制表示带符号数字的方式。整数和零的补码表示法与其二进制写法相同，只是左方要补足 0。而要表示负数，则需先将其绝对值按位取反，也就是求绝对值的一补数（也称反码），然后再加 1。例如用 8 位二进制表示数字时，5 可以表示为 0000 0101，而要表示−5，则需先求其绝对值 5，写成二进制形式 0000 0101，然后对其按位取反得到 1111 1010，再加 1。最终的结果 1111 1011就是−5 的补码表示。详情参见：https://zh.wikipedia.org/wiki/补码。带符号整数与无符号整数的英文分别为 signed integer 与 unsigned integer，其中“带符号”与“无符号”是计算机处理二进制数的两种不同方式。前者会根据最高有效位来判定数字的正负，0 为非负，1 为负，而后者则一律将其视为非负数。以上面的 1111 1011 来说，若视为带符号整数，则其值为−5（因最高位为 1，故是负数。先将其按位取反得 0000 0100，再加 1 得 0000 0101，即十进制的 5，再添上负号得−5），若视为无符号整数，则其值为 251（最高位的 1 不再表示负数，而当做 128 来算）。详情参见：https://zh.wikipedia.org/wiki/有符号数处理。——译者注

计算机算式与数学算式的书写方式相同，只是其中指代CPU寄存器中内容的变量会以粗体标出。为了和位元数组运算的通则一致，我们把电脑中的字组也视为由一串位元构成的数组。若某常量表示CPU寄存器中的值，那么它也会以粗体字出现。（这种情况在位元数组运算中找不到对应物，如果要在位元数组算式里书写常量，只能逐个列出其中的每个位元。）然而常量如果作为shift等指令的操作数，则不加粗。

对于“+”这样的操作符，若其操作数为粗体，则表明执行的是计算机加法，也就是“向量加法”，否则意味着执行的是纯数字加法。如果操作数未加粗，则其对应的操作符就是纯数学运算中的含义。要是我们想拿原来做计算机运算的粗体 **x** 值做数学运算的话，那就会把它写成不加粗的 x，其符号性应该能够从上下文中推出。假如 $\mathbf{x}=\mathbf{0x8000\ 0000}$，$\mathbf{y}=\mathbf{0x8000\ 0000}$，那么在做带符号的整数运算时，$x=y=-2^{31}$，$x+y=-2^{32}$，而 $\mathbf{x}+\mathbf{y}=\mathbf{0}$[㊀]。此处的 **0x8000 0000** 是用十六进制表示的位串，它最左边的位元是“1”，后面跟着31个“0”。

位元的序号从右侧算起，最右方的位元（也就是最低有效位）叫做0号位元。术语“位”、“半字节”、“字节”、“半字组”、“字组”、“双字”[㊁]所对应的位元数量分别是1、4、8、16、32、64。

简短的代码段用的都是计算机算式，并以左箭头表示赋值操作，偶尔还会用if语句。在这种情况下，它只是以一种与电脑平台无关的方式编写汇编语言代码罢了。

过长或过于复杂的计算机算式则用C语言来写，其代码遵循ISO 1999标准[㊂]。

完整描述C语言不是本书该做的事，不过书中用到的大部分C语言基本表达式[H & S]都总结到表1.1中了，用过程序语言编程但是不熟悉C语言的读者应该看看。表中也列出了笔者在计算机算式中用到的对应操作符，它们按照优先级从高到低的顺序排列（高者先算），在优先级这一列中，L表示左结合，比如乘号就是左结合的运算符：a·b·c=(a·b)·c，而R则表示右结合[㊃]。本书计算机算式里的运算符，其优先级与结合性同C语言一致。

表1.1 C语言与计算机算术表达式对照表

优先级	C	计算机算式	含义
	0x...	**0x...，0b...**	十六进制常数，二进制常数

㊀ 不加粗的 $x+y$ 是普通的数学运算，故结果为 -2^{32}，而粗体的 $\mathbf{x}+\mathbf{y}$ 则是按照计算机算术规则做二进制加法，其结果为1 00000000 00000000 00000000 00000000，也就是1后面32个零。宽度为32的字组无法容纳这个33位元的数，故而最高有效位的“1”会被舍去，留下32个0，因此最终的运算结果就是0。——译者注

㊁ 对应英文写法分别为：bit、nibble、byte、halfword、word、doubleword。——译者注

㊂ 该标准的正式名称是ISO/IEC 9899：1999，俗称C99，详情参见：https://zh.wikipedia.org/wiki/C语言#C99。C语言的最新标准是2011年发布的ISO/IEC 9899：2011，俗称C11。——译者注

㊃ 左结合与右结合的英文分别是left-associative与right-associative，结合性指的是遇到两个相邻的同优先级运算符时，要从左先算还是从右先算。详情参见：https://en.wikipedia.org/wiki/Operator_associativity。——译者注

（续）

优先级	C	计算机算式	含义
16	a [k]		数组 a 中索引为 k 的元素
16		x_0，x_1，…	若干个变量或位元（具体含义会在文中说明）
16	f (x, …)	f $(x, \ldots)$	求函数值
16		abs (x)	求绝对值（例外：abs (-2^{31}) $=-2^{31}$）
16		nabs (x)	绝对值对应的负数
15	x++，x--		后置自增与自减①
14	++x，--x		前置自增与自减②
14	（数据类型名）x		类型转换
14R		x^k	x 的 k 次方
14	~x	$\neg x$，$\bar{x}$	按位取反（也就是求 x 的反码）③
14	! x		逻辑非（若 x 是 0 则结果为 1，若 x 非 0 则结果为 0）
14	-x	$-x$	取相反数
13L	x * y	$x * y$	乘法，根据字组长度裁剪其运算结果④
13L	x/y	$x \div y$	带符号整数的除法
13L	x/y	$x \overset{u}{\div} y$	无符号整数的除法
13L	x % y	rem (x, y)	已知 x 与 y 为带符号的数，求 $x \div y$ 的余数。结果有可能是负数⑤
13L	x % y	remu (x, y)	已知 x 与 y 为无符号的数，求 $x \div y$ 的余数
		mod (x, y)	已知 x 与 y 为带符号的数，求 x 除以 y 的余数，并将结果调整到 [0，abs(y)−1] 之间
12L	x + y，x - y	$x+y$，$x-y$	加法与减法
11L	x<<y，x>>y	$x \ll y$，$x \overset{u}{\gg} y$	左移位，右移位（以 0 填补空缺位元，又名逻辑移位，logical shift）
11L	x>>y	$x \overset{s}{\gg} y$	右移位（根据 x 的符号来填补空缺位元，又名算术移位或数学移位）
11L		$x \overset{rot}{\ll} y$，$x \overset{rot}{\gg} y$	循环左移，循环右移
10L	x<y，x<=y，x>y，x>=y	$x<y$，$x \leqslant y$，$x>y$，$x \geqslant y$	带符号数的关系比较表达式
10L	x<y，x<=y，x>y，x>=y	$x \overset{u}{<} y$，$x \overset{u}{\leqslant} y$，$x \overset{u}{>} y$，$x \overset{u}{\geqslant} y$	无符号数的关系比较表达式
9L	x==y，x!=y	$x=y$，$x \neq y$	比较两数是否相等，比较两数是否不等
8L	x&y	$x \& y$	按位与
7L	x^y	$x \oplus y$	按位异或
7L		$x \equiv y$	按位等值，结果与 $\neg(x \oplus y)$ 相同
6L	x | y	$x \mid y$	按位或
5L	x && y	$x \overset{\rightarrow}{\&} y$	条件与（x 与 y 均不为 0 时，结果是 1，否则是 0）
4L	x || y	$x \overset{\rightarrow}{\mid} y$	条件或（x 与 y 均为 0 时，结果是 0，否则是 1）

（续）

优先级	C	计算机算式	含义
3L		$x \mid\mid y$	连接
2R	x = y	$x \leftarrow y$	赋值

① postincrement 与 postdecrement，意为先求表达式的值，然后再对其增减。假设 x 是 5，那么 x++ 的值还是 5，求完值之后，x 才变成 6。——译者注

② preincrement 与 predecrement，意为先对其增减，然后再求表达式的值。假设 x 是 5，那么先将 x 加 1，然后再判定 ++x 的值为 6。——译者注

③ one's-complement，又叫一补数，是将二进制数中的每个位元反转之后得到的数。例如 10010 的一补数为 01101。详情参见：https://zh.wikipedia.org/wiki/一补数。——译者注

④ modulo word size，意思是如果运算结果位数太多，超过了字组长度，那么会丢弃超出的部分。此概念直译为“与字长取模”或“模字长”，为了不使人误认为是和 32（字组长度值）取模，故译文用了意译，下文皆同。——译者注

⑤ 在 C99 标准中，模除的结果与被除数的正负号相同，详情参见：https://en.wikipedia.org/wiki/Modulo_operation。——译者注

除了表 1.1 列出的写法之外，书中也将借用一些布尔代数运算符与标准数学符号，在用到它们时会给出解释。

计算机算术中除了“abs”、“rem”之外，还会用到其他一些函数，等讲到那些函数时再给出其定义。

C 语言中，x<y<z 这个表达式的意思是：先判断 x<y 是否成立，若成立，则这个子表达式的值就是 1，否则为 0，然后，再将刚才的结果与 z 比较[㊀]。而在本书所讲的计算机算术中，该表达式的意思就是想判断 $x<y$ 与 $y<z$ 是否同时成立而已。

C 语言中有三种循环控制语句：while、do、for。while 语句的语法[㊁]是：

while(*expression*) *statement*

首先评估 expression 的值，若为**真**（也就是非 0），那么就执行 statement 对应的语句，然后再次判断 expression。一旦 expression 为**假**（也就是 0），while 循环就终止了。

do 语句与之相似，只是判断放在了循环尾部。它的语法是：

do *statement* while(*expression*)

先执行 statement 中的语句，然后再判断 expression。若为**真**，则继续执行循环，若为**假**，则循环终止。

for 语句的格式为：

for(e_1; e_2; e_3) *statement*

首先执行 e_1，这部分通常是赋值表达式。然后判断 e_2，它一般是个关系表达式，如果

㊀ 例如 C 语言中表达式 -5<-4<-3 就不成立，因为必须先求 -5<-4 这个子表达式的值，它成立，故被视为 1，然后再判断 1 是否小于 -3，此判断不成立，故整个表达式不成立。——译者注

㊁ 为了遵循业内惯例，语法格式中的英文单词在译文中保留原样。其中的 expression 意为“表达式”，statement 意为“语句”。——译者注

为假，则终止 for 循环，如果为**真**，那就执行 statement。最后再执行 e_3，这部分通常也是个赋值表达式。执行完毕后，又跳回 e_2 判断。因此，“do i = 1 to n” 用 for 语句写出来就是：

```
for (i=1; i<=n; i++)
```

（书中用到后置自增运算符的场合不多，这是一例。）

ISO C 标准并未规定右移（“≫”操作符）带符号的数时，左边多出来的空位是用 0 填充还是用符号填充[一]。在本书的 C 语言代码中，我们假定如果左操作数是带符号的，那么右移操作就使用符号填充，若无符号，则按 ISO 规范以 0 填充。大多数 C 编译器也都这么做。

左移操作符都采用“逻辑”填充。（某些电脑还有“算术”左移操作，也就是左移之后保持符号不变。这通常出现在老式计算机上。）

移位操作还会出现一个问题，那就是 ISO C 标准规定：假如移位的数量等于或大于左操作数的位宽，则结果未定义。虽说如此，但是几乎所有 32 位机在遇到这种情况时，都会把移位数量和 32 或 64 取模[二]。书中代码在对待此问题时，上述各种处理方式都会用到。若不同方式之间的差异关乎运算结果，则会给出解释。

1.2 指令集与执行时间模型

我们假定测试书中算法时所用的这台电脑，其 CPU 使用当前通行的 RISC 指令集，这样就好粗略估量其效率了。IBM RS/6000 系列电脑[三]的 CPU、Oracle SPARC[四]及 ARM 架构[五]的处理器都用的是 RISC。这台虚拟电脑的 CPU 使用“三段式地址”表示操作数，其寄存器相当多，至少 16 个。除非另有说明，否则寄存器都是 32 位的。0 号通用寄存器的值永远是 0，其他寄存器作用不限。

有些 CPU 中具备“特殊用途”寄存器[六]，专门保存条件比较的结果或“溢出”等状

[一] 用 0 填充（0-propagating）就是表 1.1 中所说的“逻辑”移位，而用符号填充（sign-propagating）的意思是，如果待移位的数是负值，那么就用 1 来填充，这样移位后的结果还是负的；而如果待移位的数非负，则用 0 填充，移位后的结果仍然非负。这种保持正负性的填充方式也就是上表所说的“算术”移位或“数学”移位。——译者注

[二] 举例来说，若要将一个 32 位元的数左移 80 位，那么由于 80 大于等于 32，所以必须先求它除以 32 的余数，也就是 16，然后将该数左移 16 位，得出运算结果。——译者注

[三] RS6000 是 IBM 公司使用其 RISC 架构的 Power 处理器设计生产的小型计算机。详情参见：https://zh.wikipedia.org/wiki/RS/6000。——译者注

[四] SPARC，全称为“可扩充处理器架构”（**S**calable **P**rocessor **ARC**hitecture），是 RISC 微处理器架构之一。详情参见：https://zh.wikipedia.org/wiki/SPARC。——译者注

[五] ARM 架构是一种 32 位元精简指令集（RISC）处理器架构，因其具备低成本、高效能、低耗电的特性，故广泛用于嵌入式系统。详情参见：https://zh.wikipedia.org/wiki/ARM架构。——译者注

[六] special purpose register，又叫 special function register 或 special register，详情参见：https://en.wikipedia.org/wiki/Special_function_register。——译者注

态码之类的数据，而我们为了简洁起见，假设虚拟电脑的 CPU 中没有这种寄存器。此外，它还没有浮点指令。浮点数在书中只占少量篇幅，基本上局限于第 17 章。

现在规定两套 RISC 指令集：表 1.2 是“基本 RISC 指令集”，这些指令再加上表 1.3 中的那些，就构成“完整 RISC 指令集”。

表 1.2 基本 RISC 指令集

操作码助记符	操作数	含 义
add，sub，mul，div，divu，rem，remu	RT，RA，RB	RT←RA op RB，其中 op 可以是加法、减法、乘法、带符号数的除法、无符号数的除法、带符号数的求余、无符号数的求余
addi，muli	RT，RA，I	RT←RA op I，其中 op 可以是加法或乘法，I 是 16 位元的带符号常量[①]
addis	RT，RA，I	RT←RA +（I≪16）
and，or，xor	RT，RA，RB	RT←RA op RB，其中 op 可以是按位与、按位或、按位异或
andi，ori，xori	RT，RA，Iu	同上，但最后一个操作数是 16 位元的无符号常量
beq，bne，blt，ble，bgt，bge	RT，target	在条件成立时转入 target 分支。这 6 个操作符的判断条件分别是：RT=0，RT≠0，RT<0，RT≤0，RT>0，RT≥0
bt，bf	RT，target	当 RT 是 true/false 时转入 target，与 bne/beq 等效
cmpeq，cmpne，cmplt，cmple，cmpgt，cmpge，cmpltu，cmpleu，cmpgtu，cmpgeu	RT，RA，RB	将 RA 与 RB 比较的结果存入 RT。若判断不成立，则 RT 为 0，若成立，为 1。cmpeq 表示 **compare** for **equality**（判断二者是否相等），cmpne 表示 **compare** for **in**equality（判断两者是否不等），cmplt 表示 **compare** for **l**ess **t**han（判断前者是否小于后者），以此类推。cmp 后面的字母与分支指令中的含义相同。后缀“u”表示无符号数的比较
cmpieq，cmpine，cmpilt，cmpile，cmpigt，cmpige	RT，RA，I	与 cmpeq 等 6 个操作符含义相同，只是 I 为 16 位元的带符号常量
cmpiequ，cmpineu，cmpiltu，cmpileu，cmpigtu，cmpigeu	RT，RA，Iu	与 cmpltu 等 6 个操作符含义相同，只是 I 为 16 位元的无符号常量
ldbu，ldh，ldhu，ldw	RT，d（RA）	分别将地址 RA+d 中的无符号字节、半字组、无符号半字组、字组载入 RT。d 为 16 位元的带符号常量
mulhs，mulhu	RT，RA，RB	将 RA 与 RB 之积的高 32 位存入 RT。两指令分别适用于带符号数及无符号数
not	RT，RA	将 RA 按位取反后的值存入 RT
shl，shr，shrs	RT，RA，RB	将 RA 左移位或右移位后的值存入 RT。RB 最右方的 6 个位元代表移位量。shl 与 shr 用 0 填充，shrs 用 RA 的符号位填充（移位量只取最右方 6 个位元的原因是，它需要和 64 取模，以便调整到 0 至 63 之间）
shli，shri，shrsi	RT，RA，Iu	将 RA 左移位或右移位后的值存入 RT。移位量是 Iu 所代表的 5 位元常数
stb，sth，stw	RS，d（RA）	将 RS 里的字节、半字、字组存入 RA+d 所表示的内存地址中。d 是 16 位元的带符号常量

① 本书多次出现 immediate value 一词，字面意思为“立即值”、“立即数”，它是谈到计算机指令时的一种术语，在不影响文意的情况下，译文均以“常量”、“常数”称呼之。——译者注

表 1.2、表 1.3 与表 1.4 中的 RA 与 RB，如果作为源操作数使用，就表示这些寄存器本身的内容。㊀

在实际使用的 CPU 中，尚有分支链接跳转指令（branch and link）以及返回到寄存器中所含地址的分支返回指令，前者可用来调用子程序，后者则用于从子程序中返回或实现“switch”功能。此外，可能存在一些处理专用寄存器的指令，特权指令与调用管理服务的指令当然也不少，或许还会有浮点数指令。

表 1.3 列出了 RISC 指令集中可能会用到的其他一些计算类指令，后面的章节中将会讲到它们。

汇编语言中提供了一些“扩展助记符”（Extended Mnemonic），开发人员用起来很方便，它们有点像宏，展开之后通常是一条指令。表 1.4 列举了一些可能会用到的扩展助记符。

常量加载操作会根据常量 I 的值而展开成一条或两条指令。如果 $0 \leqslant I < 2^{16}$，那么就展开成 R0 与 I 的常量按位或操作（ori 指令），而当 $-2^{15} \leqslant I < 0$ 时，则会展开成 R0 与 I 的常量加法操作（addi 指令）。若 I 最右方的 16 个位元都是 0，那么就视为常量移位加法操作（addis 指令），而在不属于上述三种情况时，则会展开成一条 addis 与一条 ori 指令。㊁（在最后一种情况下，也可以直接使用从内存中加载数值的指令来做。但是为了便于估量算法所需的执行时间和空间，我们假定此种常量加载操作总是相当于两个算术指令。）

表 1.3 “完整 RISC 指令集”中的其余附加指令

操作码助记符	操作数	含 义
abs，nabs	RT，RA	将 RA 的绝对值或“负绝对值”① 存入 RT
andc，eqv，nand，nor，orc	RT，RA，RB	将 RB 按位取反后与 RA 按位与，按位比较，按位与后按位取反，按位或后按位取反，将 RB 按位取反后与 RA 按位或
extr	RT，RA，I，L	将 RA 中序号为 I 至 I+L−1 之间的位元提取出来，靠右存放在 RT 中，余位填 0
extrs	RT，RA，I，L	与 extr 相同，但是用符号填充
ins	RT，RA，I，L	将 RA 中序号为 0 至 L−1 的位元插入 RT 中序号为 I 至 I+L−1 的位置上
nlz	RT，RA	取得 RA 中前导 0 的个数（该操作的结果在 0 至 32 之间）
pop	RT，RA	RT 计算 RA 中有多少个值为 1 的位元（该操作的结果在 0 至 32 之间）
ldb	RT，d（RA）	从 RA+d 所在的内存地址处加载一个带符号的字节到 RT 中。d 为 16 位元的带符号常量
moveq，movne，movlt，movle，movgt，movge	RT，RA，RB	如果 RA 与 0 的关系满足判断条件，就将 RB 赋值给 RT，否则 RT 的值不变。eq 与 ne 分别表示 RA=0、RA≠0，其余依此类推
shlr，shrr	RT，RA，RB	将 RA 循环左移位或循环右移位。RB 最右方的 5 个位元表示移位的数量

㊀ 若 RA、RB 不以带括号形式出现，即表示其本身值，否则视为定位所用的内存地址。——译者注

㊁ 常量加载操作要分成三种情况展开的原因，请参考：http://www.engr.uconn.edu/~jeffm/Classes/CSE240-Spring-2000/Lectures/lecture6/node4.html。——译者注

（续）

操作码助记符	操作数	含 义
shlri，shrri	RT，RA，Iu	将 RA 循环左移位或者循环右移位，移位的数量在 5 位立即数字段中给出
trpeq，trpne，trplt，trple，trpgt，trpge，trpltu，trpleu，trpgtu，trpgeu	RA，RB	当两个操作数满足 RA = RB、RA≠RB 等条件时，令处理器中断（trap）
trpieq，trpine，trpilt，trpile，trpigt，trpige	RA，I	与 trpeq 等指令含义相同，只是第 2 个操作数为 16 位元的带符号常量
trpiequ，trpineu，trpiltu，trpileu，trpigtu，trpigeu	RA，Iu	与 trpltu 等指令含义相同，只是第 2 个操作数为 16 位元的无符号常量

① negative of the absolute value 是一种特有的绝对值解读方式，详情参见：http：//pic.dhe.ibm.com/infocenter/aix/v7r1/index.jsp? topic=%2Fcom.ibm.aix.aixassem%2Fdoc%2Falangref%2Fidalangref_nabs_negabs_instrs.htm。——译者注

表 1.4 扩展助记符

扩展助记符	展开式	含 义
b target	beq R0，target	无条件执行分支
li RT，I	详见书中解释	将常量加载至 RT 中。$-2^{31} \leqslant I < 2^{32}$
mov RT，RA	ori RT，RA，0	将寄存器 RA 中的值赋给 RT
neg RT，RA	sub RT，R0，RA	将 RA 的相反数（也就是二补码）放入 RT
subi RT，RA，I	addi RT，RA，-I	将 RA 减 I 的差放入 RT（$I \neq -2^{15}$）①

① 因为 -2^{15} 的相反数是 2^{15}，该值无法以 16 个位元容纳，所以在这种特殊情况下，subi 与 addi 展开式不等效。——译者注

一个指令到底应该属于基本 RISC 指令集还是完整 RISC 指令集，其实取决于个人的解读。有人可能会觉得无符号数除法与求余操作应该放在完整 RISC 指令集中，而与此相反，带符号的字节加载（load byte signed，ldb）操作反而应该归入基本 RISC 指令集。ldb 操作之所以会被放在完整指令集中，是因为其使用频率相当低，而且该指令需要将符号位扩展很多次㊀，这样做比较耗费 CPU 周期。

划分基本 RISC 指令集与完整 RISC 指令集时，还有许多可以商榷的地方，此处不再赘言。

书中所用指令最多只会用到两个源寄存器与一个目标寄存器，这样做也简化了计算机的 CPU 架构（比方说，寄存器堆㊁中最多只需两个读端口与一个写端口即可）。此外也缓解了优化编译㊂的工作量，使它不用再处理那些用到多个目标寄存器的指令了。这样做

㊀ 如果待加载字节的最高有效位是 1，则表明其为负数，将它加载到一个 32 位宽的寄存器时，必须把多出来的 24 个空位元全部扩展为 1，这样才能保证寄存器中的值和正负性与原字节相符。——译者注

㊁ Register File，是 CPU 中多个寄存器组成的阵列，通常由快速的静态随机读写存储器（SRAM）实现。这种 RAM 具有专门的读端口与写端口，可以多路并发访问不同的寄存器。详情参见：https：//zh.wikipedia.org/wiki/寄存器堆。——译者注

㊂ Optimizing Compiler，是优化程序所用的编译器，可以加快程序执行速度或减低内存占用量等。详情参见：https：//en.wikipedia.org/wiki/Optimizing_compiler。——译者注

的代价是，有些本来执行一条指令就能实现的操作现在得分解成两条。比如某些程序想同时知道两个数的商和余数（这种情况很罕见），那么现在就必须用除法和求余两个操作才能完成。在现实环境下的CPU中，余数是除法操作的副产品（by-product），很多电脑在执行除法指令时顺便计算好了余数。要获取保存两个字组乘积的双字时，也会遇到这个问题。

条件移动指令（conditional move，例如moveq）表面上看只有两个源操作数，但换个角度看，也可以说是3个。因为该指令的结果取决于RT、RA与RB的值，所以乱序执行的CPU必须把这种指令里的RT标注为use和set。也就是说，如果前面一条指令修改了RT，而后面紧跟一个将要再度修改RT的条件移动指令，那么就只能按照这个顺序执行，同时不能丢弃前一条指令的运算结果。有鉴于此，此类处理器的设计者可以选择略过条件移动操作以避免处理这种（从逻辑上讲）需要3个源操作数的指令。反之，使用条件移动指令确实能简化分支的实现。

指令格式与本书内容无关，不过上面列出的所有RISC指令集，再加上浮点指令和一些管理用的指令，都可以在有32个通用寄存器的电脑上用32个位元（其中用于指示寄存器序号的字段占5个位元㊀）加以实现。如果将compare、load、store、trap指令所支持的常数字段缩减为14位，那么这些指令在具备64个通用寄存器的电脑上，也照样能用32个位元做出来（其中用于指示寄存器序号的字段占6个位元㊁）。

执行时间

本书假设所有指令都只需1个CPU周期即可执行完毕，但是乘法、除法及求余指令例外，我们不去预估这3个指令的执行时间。而分支指令不论其执行结果是转入分支还是继续沿主线执行，都只花1个周期。

常量加载指令会花费一至两个周期，因为要把常数存入寄存器中，所需的算术指令数可能是1个，也可能是两个。

书里用到加载与存储指令的场合不多，我们也假设其只需1个周期，而且忽略加载延迟（也就是从算术逻辑单元㊂执行完加载指令算起，到待加载的数据真正可以用于后续指令为止，这两者的时间间隔）。

然而，只晓得算术与逻辑指令的执行周期，通常无法估算出程序执行时间，因为数

㊀ 在CPU中，除了操作数之外，指令本身也是用二进制来表示的。在由32个位元构成的指令中，一般会有指示操作类型的操作码字段、标识寄存器序号的字段，以及容纳常量的字段等。5个位元能表达值为0～31的32个数，所以刚好可以对应32个通用寄存器的序号。详情参见：https：//zh.wikipedia.org/wiki/指令集架构。——译者注

㊁ 一般常量用16个位元表示，而指令码本身通常含有两个表示寄存器的字段，现在要将其由5位扩展至6位，所以原来表示常量用的字段就会缩减为14位。——译者注

㊂ 原文为arithmetic unit，应是arithmetic logic unit（ALU）的简称，它是中央处理器的执行单元，也是所有CPU的核心组成部分，由“And Gate”和“Or Gate”构成，主要用于进行加、减、乘等二进制算术运算。详情参见：https：//zh.wikipedia.org/wiki/算术逻辑单元。——译者注

据加载延迟及获取指令延迟可能会大大减缓程序执行速度。尽管近些年人们越来越重视这个问题了，但是本书不打算讨论。另外还有一个因素会改善执行效率，那就是所谓的"指令级并发"(instruction-level parallelism)。很多时下流行的 RISC 芯片都支持该技术，那些"高端"(high-end) 电脑尤其如此。

此类机器的 CPU 都有多个执行单元，并具备足够的指令派发能力，以便并发执行互不依赖的一组指令（也就是说，这些指令的执行结果都不取决于其他指令，而且不会同时修改某个寄存器或状态位）。由于此技术已非常普遍，所以本书会标明那些相互独立的指令。如此一来，我们就可以说：某某公式用 8 条指令即可实现，并且在一台具备无限指令级并发能力的电脑上运行，只需 5 个周期。这也意味着，假使 CPU 能够合理安排(schedule) 各条指令的执行顺序，而且其中加法器[㊀]、移位器 (shifter)、逻辑单元及寄存器的数量足够多，那么从理论上讲，只要 5 个周期就能执行完这段代码了。

然而也不能过分强调这一点，因为不同电脑的指令级并行能力差得很远。比如一台 1992 年加州产的 IBM RS/6000 系列电脑[㊁]，其 CPU 的加法器就有 3 个输入端，甚至能够并发执行两条首位衔接的加法类指令（比如加法指令的目标寄存器当做比较指令或加载指令的源寄存器用[㊂]）。与之相对，那种在低端嵌入式设备上运行应用程序所用的 CPU 的构造非常简单，其寄存器堆栈可能只有一个读端口。一般说来，这种机器执行需要两个源寄存器的指令时，会多花一个周期来读取第 2 个操作数。然而这种 CPU 也许会有一个旁路 (bypass)，当某条指令的目标寄存器恰好是下面那条指令的源操作数时，它不用再通过寄存器堆的读端口就可以把寄存器拿来用。在这种设备上以非并行方式执行首位衔接的代码反倒发挥了它的优势。

1.3 习题

1. 将如下循环用 while 方式改写：

   for $(e_1;\ e_2;\ e_3)$ *statement*

   此循环能用 do 表示吗？
2. 用 C 语言写一个循环，控制名为 i 的无符号整数变量，使其值从 0 开始，递增到 32 位机所能表示的最大无符号数 0xFFFF FFFF（循环范围包含这两个极值）。
3. 此题留给知识丰富的读者：本书所列的基本 RISC 指令集和完整 RISC 指令集中，每条指令最多只需读取两次寄存器，写入一次寄存器。请问在常见的 RISC 指令中，有没有那种貌似简单但实际上却需要操作两个以上源数据，或多次写入寄存器的指令呢？

㊀ 在电子学中，加法器 (adder) 是一种用于执行加法运算的数字电路部件，是计算机核心微处理器中算术逻辑单元的基础。在这些电子系统中，它主要负责计算地址、索引等数据。此外，它还是二进制乘法器等其他硬件的重要组成部分。详情参见：https://zh.wikipedia.org/wiki/加法器。——译者注

㊁ IBM RS/6000 系列电脑的发展历程可参阅其官方文档：http://www-03.ibm.com/ibm/history/documents/pdf/rs6000.pdf。——译者注

㊂ 原书的说法是 one feeds the other，即前一条指令将操作结果作为数据源，"喂"给后一条指令。——译者注

第 2 章

基础知识

2.1 操作最右边的位元

本节所讲的公式有一些可能会在后面章节中用到。

下列公式可以将字组中值为 1 且最靠右的位元“关闭”[1]，如果不存在值为 1 的位元，那结果就是 0（例如 0101 1110⇒0101 0000）：

$$\boldsymbol{x} \ \&(\boldsymbol{x}-\mathbf{1})$$

该操作可判断某个无符号整数是不是 2 的幂或 0：套用此公式后判断运行结果是否为 0 即可。

下列公式可以将字组中值为 0 且最靠右的位元“打开”[2]，如果不存在值为 0 的位元，则结果的每一位都是 1（例如 1010 0111⇒1010 1111）：

$$\boldsymbol{x} \mid (\boldsymbol{x}+\mathbf{1})$$

下面的公式可以将字组尾部的 1 都变成 0，如果尾部没有 1，则 x 不变（例如 1010 0111⇒1010 0000）：

$$\boldsymbol{x} \ \&(\boldsymbol{x}+\mathbf{1})$$

套用此公式后判断结果是否为 0，即可确定某个无符号整数是不是 2^n-1 或 0，也可以判断某个数的所有位元是否均为 1。

以下公式可以将字组尾部的 0 都变成 1，如果尾部没有 0，则 x 不变（例如 1010 1000⇒1010 1111）：

$$\boldsymbol{x} \mid (\boldsymbol{x}-\mathbf{1})$$

下面的公式可以把 x 中最靠右且值为 0 的位元变为 1，并将其余位元置 0。如果 x 中没有值为 0 的位元，则结果为 0（例如 1010 0111⇒0000 1000）：

$$\neg\boldsymbol{x} \ \&(\boldsymbol{x}+\mathbf{1})$$

下面的公式可以把 x 中最靠右且值为 1 的位元变成 0，并将其余位元置 1。如果 x 中

[1] 原文为 turn off，也就是将位元的值变为 0。——译者注

[2] 原文为 turn on，也就是将位元的值变为 1。——译者注

没有值为 1 的位元，则运算结果中的每个位元均是 1（例如 1010 1000⇒1111 0111）：

$$\neg x \mid (x-1)$$

下列 3 个公式都可以把字组尾部所有值为 0 的位元变成 1，并将其余位元置 0。如果 x 中没有值为 0 的位元，则结果是 0（例如 0101 1000⇒0000 0111）：

$$\neg x \,\&(x-1)，或$$

$$\neg(x \mid -x)，或$$

$$(x \,\&-x)-1$$

第 1 个公式可以发挥处理器的某些指令级并行能力。

下列公式可以把字组尾部所有值为 1 的位元都变成 0，并将其余位元设为 1。如果 x 中没有值为 1 的位元，则运算结果中的每个位元均是 1（例如 1010 0111⇒1111 1000）：

$$\neg x \mid (x+1)$$

下列公式可以保留字组中最靠右且值为 1 的位元，并将其余位元置 0。若 x 不存在值为 1 的位元，则运算结果为 0（例如 0101 1000⇒0000 1000）：

$$x \,\&(-x)$$

下列公式可以将字组最靠右且值为 1 的位元，及其右方所有值为 0 的位元都变成 1，并将左方位元置 0。若 x 中没有值为 1 的位元，则运算结果的每一位都是 1，而当 x 尾部没有值为 0 的位元时，运算结果是 1（例如 0101 1000⇒0000 1111）：

$$x \oplus (x-1)$$

下列公式可以将字组最靠右且值为 0 的位元，及其右方所有值为 1 的位元都设为 1，并将左方位元置 0。若 x 中没有值为 0 的位元，则运算结果的每一位都是 1，而当 x 尾部没有值为 1 的位元时，运算结果是 1（例如 0101 0111⇒0000 1111）：

$$x \oplus (x+1)$$

下列两个公式都可以把字组右侧连续出现且值为 1 的位元置 0（例如 0101 1100⇒0100 0000）[Wood]：

$$(((x \mid (x-1))+1)\,\&\, x)，或$$

$$((x \,\&-x)+x)\,\&\, x$$

如果 x 是非负数，而且套用上述公式运算的结果是 0，那就表明它可以写为 2^j-2^k 的形式。其中 $j \geqslant k \geqslant 0$。

2.1.1 德摩根定律的推论

描述德摩根定律[⊖]的两个逻辑恒等式可以理解为把取反符号（not sign）“分配”到每一个变量中。结合下列等式（头两个就是德摩根定律）与德摩根定律的思路，可以为本节所讲的公式及其他一些公式变形。

⊖ 德摩根定律（De Morgan's law）是由英国数学家、逻辑学家奥古斯塔斯·德摩根（Augustus De Morgan，1806—1871）所陈述的逻辑学定律。详情参见：https：//zh.wikipedia.org/wiki/德摩根定律。——译者注

$$\neg(x \,\&\, y) = \neg x \mid \neg y$$
$$\neg(x \mid y) = \neg x \,\&\, \neg y$$
$$\neg(x+1) = \neg x - 1$$
$$\neg(x-1) = \neg x + 1$$
$$\neg -x = x - 1$$
$$\neg(x \oplus y) = \neg x \oplus y = x \equiv y$$
$$\neg(x \equiv y) = \neg x \equiv y = x \oplus y$$
$$\neg(x+y) = \neg x - y$$
$$\neg(x-y) = \neg x + y$$

上述公式可以这样用：$\neg(x \mid -(x+1)) = \neg x \,\&\, \neg -(x+1) = -\neg x \,\&\, ((x+1)-1) = \neg x \,\&\, x = 0$。

2.1.2 从右至左的可计算性测试

有一种简单的办法可以判断出某个函数是否能通过一系列加法、减法、按位和、按位与及按位取反实现出来[㊀][War]。当然，还有一些能够通过上述基本操作组合而成的运算方法也在允许范围内，比如移位数量固定的左移操作（等价于一系列加法操作）或乘法操作等。除此之外的其余操作都是不允许的。下面这条定理描述了如何判断一个函数是否可以按照从右至左的顺序实现出来。

定理 一个从字组到字组的映射函数，当且仅当运算结果中的每一个位元只依赖于各操作数中对应位元及其右侧位元时，才可以用一系列字并行[㊁]加减法、按位与、按位或及按位取反操作实现出来。[㊂]

也就是说，要判断某个函数是否能够从右至左实现出来，大家需要设想：运算结果最右侧的那个位元能不能只用各个操作数最右侧的位元来算，而运算结果的倒数第 2 个位元能不能只能用各操作数最右方的两个位元来算，以此类推。如果能这么算出来，那就说明该函数可以用一系列的加法、按位与等简单操作实现出来。若是不能，那就表示只通过上述基本指令无法实现这个函数。

定理中有意思的地方就是“当且仅当……”这个分句。按照定理中的说法，加、减、按位与、按位或、按位取反等操作都能按照从右至左的顺序实现出来，于是，由这些操作组合而成的各种复合运算方式居然也能从右至左算出来，这实在出乎很多人意料。

㊀ 也就是判断某个函数是否具备本节标题所谓的“从右至左的可计算性”（right-to-left computability test）。——译者注

㊁ word-parallel，是将每个字组中的位元横列出来，再把若干个字组纵向对齐写出的一种演算方式。——译者注

㊂ 该定理用通俗的话来表述就是：如果一个函数能够以位元为单位从右至左计算出来，那么它就肯定能用那 5 个简单操作及其复合指令实现；反之，如果能用这些基本指令来描述一个函数，那么它必然也能通过从右至左的按位运算方式做出来。——译者注

为了验证定理中的“当且仅当……”这句话，需要用一点笨办法才行。现在构建如下特例：假设有一个函数具备从右至左的可计算性，它有两个名为 x、y 的变量，而且2号位元 r 是用如下算式得出的：

$$r_2 = x_2 \mid (x_0 \ \& \ y_1) \tag{1}$$

按照从右至左的顺序，将各个位元标为0至31号。由于计算2号位元时只用到了各操作数中序号小于等于2的位元，所以2号位元就可以“从右至左计算出来”（right-to-left computable）。

把字组 $\boldsymbol{x}$、$\boldsymbol{x}$ 左移两位后的数、$\boldsymbol{y}$ 左移一位后的数从上至下写出来，再写一个只有2号位为1的掩码。

$$\begin{array}{ccccccc}
x_{31} & x_{30} & \cdots & x_3 & x_2 & x_1 & x_0 \\
x_{29} & x_{28} & \cdots & x_1 & x_0 & 0 & 0 \\
y_{30} & y_{29} & \cdots & y_2 & y_1 & y_0 & 0 \\
0 & 0 & \cdots & 0 & 1 & 0 & 0 \\
0 & 0 & \cdots & 0 & r_2 & 0 & 0
\end{array}$$

将字组横列式中第2行与第3行按位与，然后和第1行按位或（操作的依据是等式(1)），最后跟第4行的掩码按位与。这么计算的结果是，除了要计算的2号位元外，其余位置都变成了0。对于待求值的其他31个位元，也都按照此方法演算，最后把32个字组按位或，于是就实现了整个函数。

按照此方式构建的算法效率并不高，不过这里只是为了演示此类函数的确能用5种基本指令拟合出来而已。

根据上述定理，立刻就能断定：无法通过5种基本指令序列把字组最左侧且值为1的位元置0，因为要想判断字组中某个值为1的位元该不该变成0，必须继续向左方搜寻，以确保其左侧再也没有其他值为1的位元了。同理，右移、循环移位、移位长度为变量的左移、计算字组尾部值为0的位元个数等操作都无法用5种简单的运算组合出来（在计算尾部值为0的位元个数时，如果结果为奇数，那么其最右侧的位元必然是1，而要想确定这一点，就必须继续向左看，以确保操作数尾部真的有奇数个值为0的位元）[⊖]。

2.1.3 位操作的新式用法

位操作还有一种新用法，也用到了上面那些技巧，那就是给定一个数，然后找出下一个比它大而且值为1的位元数与之相同的数字。读者可能会问：计算这样的数有什么用

⊖ 作者的意思是，假如要从右至左算出0100 1000这个数尾部有多少个0（正确答案是3个，用二进制写出则是0000 0011个），必须先确定答案的最右方位元是0还是1，而这将由操作数尾部0的个数是奇数还是偶数来决定，要想确定这个事实，却必须继续向左看，才能知道尾部的0到底是奇数个、还是偶数个。这样一来，就违背了“从右至左”的运算规则——计算答案最右方的位元时，只能用操作数最右方的位元来算，不能向左取值。——译者注

呢？它用在以位串来表示子集的领域。集合中的元素可以写成一行，然后可以用一个字组或字组序列来表示它的子集。如果集合中的某个元素在子集中，那么字组里对应的位元就为 1，否则为 0。想求两个子集的并集，只需要把对应的位串按位取或即可，而想求交集，就按位取和，此外还有很多类似的操作。

有时可能需要找出元素个数为某一定值的全部子集。如果有一个函数，能够根据某个给定的整数（将表示子集的位串视为整数），找出下一个比它大而值为 1 的位元个数又与之相同的数，那么很快就能据此找出所有子集了。

R. W. Gosper［HAK，item 175］设计出了此操作的精确算法[㊀]。给定一个表示子集位串的字组 x，首先要找到连续出现在 x 右侧且值为 1 的一组位元，然后将该值“加 1”，再把原来后面跟着的那些 0 补上。举例来说，如果待计算的位串是 xxx0 1111 0000，那么结果就应该是 xxx1 0000 0111，其中 xxx 这三个位元的值不限。该算法首先定义 $s = x \,\&\, -x$，并算出 s 等于 0000 0001 0000，这样就找到了 x 中“最小”[㊁]的那个 1。然后把它与 x 相加，把两数之和 xxx1 0000 0000 保存在 r 里。此时结果中的一个位元已经算好了，它就是 r 里面的那个“1”。想求出其他位元，还需要把位串中剩下的 $n-1$ 个“1”移到右侧（其中 n 指的是连续出现在 x 右侧且值为 1 的位元个数）。要把它们移到右边，首先得计算 r 和 x 的异或值，在本例中就是 0001 1111 0000。

上面那个值中“1”的个数太多了，而且没有靠右对齐。为解决此问题，要将它与 s 相除，这样就可以把那些“1”靠右对齐了（因为 s 是 2 的幂），除之前还要先向右移两位，以便丢弃那两个多余的位元。将此结果与 r 取或，就得到最终答案了。

最终结果 y 用计算机代数来表示，就是：

$$
\begin{aligned}
s &\leftarrow x \,\&\, -x \\
r &\leftarrow s + x \\
y &\leftarrow r \mid (((x \oplus r) \overset{u}{\gg} 2) \overset{u}{\div} s)
\end{aligned}
\tag{2}
$$

图 2.1 中列出了此算法的完整 C 语言实现，它执行了 7 个基本的 RISC 指令，其中 1 个用于除法。（不要以 0 为参数调用此段程序，否则会因与 0 相除而出错。）

```
unsigned snoob(unsigned x) {
   unsigned smallest, ripple, ones;
                                // x = xxx0 1111 0000
   smallest = x & -x;           //     0000 0001 0000
   ripple = x + smallest;       //     xxx1 0000 0000
   ones = x ^ ripple;           //     0001 1111 0000
   ones = (ones >> 2)/smallest; //     0000 0000 0111
   return ripple | ones;        //     xxx1 0000 0111
}
```

图 2.1 计算下一个值比 x 大而 1 的个数与之相同的数

若是嫌除法太慢，可以用另外一种算法代替。设函数 ntz(x) 为 x 尾部 0 的个数，nlz

㊀ 该算法的另外一种形式请参考［H &S］7.6.7 节。

㊁ 也就是最靠右且值为 1 的位元。——译者注

(x) 为 x 头部 0 的个数，而 pop(x) 为 x 中 1 的个数（又叫种群统计函数，population count），假如能够快速计算出这三个函数的话，那么就可以用下列三个公式之一来取代等式（2）的最后一个式子。（如果公式中的右移操作用的是“模 32 移位”[⊖]，那么前两种替代方案无法得出正确结果。）

$$y \leftarrow r \mid ((x \oplus r) \overset{u}{\gg} (2 + \mathrm{ntz}(x)))$$

$$y \leftarrow r \mid ((x \oplus r) \overset{u}{\gg} (33 - \mathrm{nlz}(s)))$$

$$y \leftarrow r \mid ((1 \ll (\mathrm{pop}(x \oplus r) - 2)) - 1)$$

2.2 结合逻辑操作的加减运算

本书假定读者已经熟悉普通代数运算及布尔运算中的一些基本恒等式。下面列出一组将逻辑操作与加减法结合起来的恒等式。

a. $-x = \neg x + 1$

b. $= \neg(x - 1)$

c. $\neg x = -x - 1$

d. $-\neg x = x + 1$

e. $\neg -x = x - 1$

f. $x + y = x - \neg y - 1$

g. $= (x \oplus y) + 2(x \,\&\, y)$

h. $= (x \mid y) + (x \,\&\, y)$

i. $= 2(x \mid y) - (x \oplus y)$

j. $x - y = x + \neg y + 1$

k. $= (x \oplus y) - 2(\neg x \,\&\, y)$

l. $= (x \,\&\, \neg y) - (\neg x \,\&\, y)$

m. $= 2(x \,\&\, -y) - (x \oplus y)$

n. $x \oplus y = (x \mid y) - (x \,\&\, y)$

o. $x \,\&\, \neg y = (x \mid y) - y$

p. $= x - (x \,\&\, y)$

q. $\neg(x - y) = y - x - 1$

r. $= \neg x + y$

s. $x \equiv y = (x \,\&\, y) - (x \mid y) - 1$

t. $= (x \,\&\, y) + \neg(x \mid y)$

⊖ modulo 32 shift，CPU 在执行此指令前，先要把移位的个数（也就是该操作符的右操作数）调整成一个大于等于 0 且小于 32 的整数，故有此称呼。——译者注

$$\text{u.}\qquad x \mid y = (x \,\&\, \neg y) + y$$

$$\text{v.}\qquad x \,\&\, y = (\neg x \mid y) - \neg x$$

等式（d）可以重复运用，例如$-\neg-\neg x=x+2$。同理，多次套用等式（e），也可以推出$\neg-\neg-x=x-2$。由此可见，与常量的加减法都可以只用按位取反及求相反数操作组合而成[⊖]。

等式（f）与（j）对偶。大家都知道：利用等式（j）所描述的关系，可以通过加法器构造出减法器。

等式（g）和（h）记载于 HAKMEM 备忘录中［HAK，item23］。等式（g）先对两数做不进位加法（$x \oplus y$），然后再补上进位。等式（h）改变了加法运算的顺序，使得在计算任何一个位元的过程中，都不可能出现 0+1 的组合。原来会出现此情况的地方，现在都变成了 1+0。

在普通的二进制加法中，若单独分析运算结果中的每一个位元，则它等于 0 或等于 1 的概率相等，而在每个位置上发生进位的概率则是 0.5。但是如果在制作加法器时，使用等式（g）描述的这种方法对输入数字进行预处理的话，那么进位概率就降到 0.25 了。这一论述对构建加法器的意义不是太大，因为设计加法器时关心的是，能不能把进位在逻辑电路中传播的最大位元数降到最低，而等式（g）只能把这个传播距离缩小 1 位。

用于减法操作的等式（k）、（l）分别与加法操作中的（g）、（h）对偶。也就是说（k）先对两数做不进位减法（$x \oplus y$），然后再把借来的位从结果中减去。同理，等式（l）只不过调整了操作数，使得在计算任何一个位元的过程中，都不可能出现 1－1 这样的组合。原来会出现此情况的地方，现在都变成了 0－0。

等式（n）演示了只需 3 个基本 RISC 指令就可实现异或操作。如果只用与、或、非这 3 种逻辑操作来实现，则需 4 个指令：$((x \mid y) \,\&\, \neg(x \,\&\, y))$。同理，等式（u）和（v）演示了如何用 3 个基本的指令来实现按位与及按位或操作。这两个操作如果用德摩根定律的形式来做，需要 4 个指令。

2.3 逻辑与算术表达式中的不等式

如果将二元逻辑表达式[⊜]中的值视为无符号整数，则很容易推出一系列不等式来。比如下面这两个例子：

⊖ 原文为“using only two forms of complementation”，意为“只用两种形式的补码就可以算出”。因为按位取反与求相反数操作分别相当于求某数的“一补码”（俗称“反码”）和“二补码”（俗称“补码”），故有此说。——译者注

⊜ binary logical expression，也可以理解为“二进制逻辑表达式”，因为操作数是 0、1 这两个二进制数，然而 binary 一词更多是在强调这些逻辑运算的操作数多为两个，而且其值和最后的运算结果也是以“真”、“假”两种形态出现的，故译文在不致混淆时，均将其称为“二元逻辑表达式”。——译者注

$$(\boldsymbol{x} \oplus \boldsymbol{y}) \overset{u}{\leqslant} (\boldsymbol{x} \mid \boldsymbol{y}) \text{ 和}$$

$$(\boldsymbol{x} \,\&\, \boldsymbol{y}) \overset{u}{\leqslant} (\boldsymbol{x} \equiv \boldsymbol{y})$$

表 2.1 列出了所有二元逻辑操作的运算结果。上述二式皆可由此得出。

表 2.1　16 种二元逻辑操作

x	y	0	$x \& y$	$x \& -y$	x	$\neg x \& y$	y	$x \oplus y$	$x \mid y$	$\neg(x \mid y)$	$x \equiv y$	$\neg y$	$x \mid \neg y$	$\neg x$	$\neg x \mid y$	$\neg(x \& y)$	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

设 $f(x,y)$ 与 $g(x,y)$ 为表 2.1 中的两列，如果当每一行的 $f(x,y)$ 值为 1 时，对应的 $g(x,y)$ 也是 1，那么就可以说不等式 $f(x,y) \leqslant g(x,y)$ 对于所有的 (x,y) 都成立。这一规律显然也能推广至字组并列式的各种逻辑操作上。大部分此类不等式都是很容易就能推算出来的，像 $(x \,\&\, y) \overset{u}{\leqslant} x \overset{u}{\leqslant} (x \mid \neg y)$ 等。此外，如果发现某一行所对应的两列中，前者是 0，后者是 1，而在另外一行中，前者是 1，后者是 0，那么对应的逻辑表达式就不存在不等关系。用上面这些知识，很容易就能判断出不等式 $f(x,y) \overset{u}{\leqslant} g(x,y)$ 对于每一组二元逻辑函数 f 与 g 是否成立了。

使用这些关系式时要注意：在普通算术中，如果 $x+y \leqslant a$ 且 $z \leqslant x$，那么 $z+y \leqslant a$，而若将上述不等式中的“+”换成了或运算，这种递推关系就不再成立了。

同时含有逻辑表达式与算术表达式的不等式更有意思，选几个例子列在下面。

a.　$(\boldsymbol{x} \mid \boldsymbol{y}) \overset{u}{\geqslant} \max(\boldsymbol{x}, \boldsymbol{y})$

b.　$(\boldsymbol{x} \,\&\, \boldsymbol{y}) \overset{u}{\leqslant} \min(\boldsymbol{x}, \boldsymbol{y})$

c.　$(\boldsymbol{x} \mid \boldsymbol{y}) \overset{u}{\leqslant} \boldsymbol{x} + \boldsymbol{y}$　若加法操作未溢出

d.　$(\boldsymbol{x} \mid \boldsymbol{y}) \overset{u}{>} \boldsymbol{x} + \boldsymbol{y}$　若加法操作溢出

e.　$|\boldsymbol{x} - \boldsymbol{y}| \overset{u}{\leqslant} (\boldsymbol{x} \oplus \boldsymbol{y})$

可能除了 $|\boldsymbol{x} - \boldsymbol{y}| \overset{u}{\leqslant} (\boldsymbol{x} \oplus \boldsymbol{y})$ 这个式子外，其余几个都不难证明。$|\boldsymbol{x} - \boldsymbol{y}|$ 的意思就是取 $\boldsymbol{x} - \boldsymbol{y}$ 的绝对值，在无符号数的范围内，该值与算式 $\max(\boldsymbol{x}, \boldsymbol{y}) - \min(\boldsymbol{x}, \boldsymbol{y})$ 相等。要想证明这一点，可以对 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 的位数运用数学归纳法（证明其中的递推关系式时，展开等式左侧会比较容易些）。

2.4　绝对值函数

要是你所用的 CPU 没有计算绝对值的指令，那可以考虑用下面几种方式来计算。这

些算法通常用三、四条指令就能实现，而且不带分支。首先执行 $y \leftarrow x \overset{s}{\gg} 31$，然后套用下列式子之一即可：

$$\begin{array}{cc} \text{abs} & \text{nabs} \\ (x \oplus y) - y & y - (x \oplus y) \\ (x + y) \oplus y & (y - x) \oplus y \\ x - (2x \,\&\, y) & (2x \,\&\, y) - x \end{array}$$

这里的 $2x$ 指的就是 $x + x$ 或 $x \ll 1$。

若 CPU 能快速算出一个数与±1 的乘积，那么可以考虑用这个式子求绝对值：

$$((x \overset{s}{\gg} 30) \mid 1) * x$$

2.5　两数平均值

下列公式［Dietz］可以算出两个无符号数的平均值 $\lfloor (x+y)/2 \rfloor$，而且还不会溢出：

$$(x \,\&\, y) + ((x \oplus y) \overset{u}{\gg} 1) \tag{3}$$

而下式则可算出无符号整数式 $\lceil (x+y)/2 \rceil$ 的值：

$$(x \mid y) - ((x \oplus y) \overset{u}{\gg} 1)$$

若要分别计算带符号整数的两种平均值（一种是出现小数时向下取整，另一种是向上取整），只需把公式中的无符号移位操作换成带符号移位即可。

对于两个带符号的整数，有时可能要计算向 0 取整之后的平均值。想求这种“截尾平均值”[⊖]（在不溢出的情况下）稍微有点难。可以先算出向下取整之后的平均值，然后再修正。修正它的办法是：如果 $x + y$ 的算术值是负奇数，则将结果再加 1。然而当且仅当 (3) 式运算结果为负时（其中的无符号移位变成带符号移位），$x + y$ 才会是负数。综上所述，可以得出如下计算方式（在归并了重复的子表达式 $x \oplus y$ 之后，只需 7 个基本 RISC 指令即可）：

$$t \leftarrow (x \,\&\, y) + ((x \oplus y) \overset{s}{\gg} 1);$$

$$t + ((t \overset{u}{\gg} 31) \,\&\, (x \oplus y))$$

有些特例可以用更快的办法算出来。比如假设 x 和 y 都是带符号的非负数，那么只需要算出 $(x + y) \overset{u}{\gg} 1$ 就能知道其平均值了。虽然两数之和可能会溢出，但是溢出的那一

⊖ truncated average，也就是向 0 取整之后的平均值。其含义是：若平均值是小数，则调整为距离该值最近且更靠近 0 的那个整数。例如 3 和 −8 取平均值，结果是 −2.5，−3 与 −2 这两个整数和它的距离都是 0.5，然而 −2 更靠近 0，所以 3 与 −5 的“截尾平均值”是 −2，而不是 −3。——译者注

位[一]仍然保留在存放加法结果的寄存器里，所以只需要执行一次无符号右移，就可以把这个溢出位挪到正确的地方，并且让空出来的那个符号位填上 0。

如果 x 和 y 都是无符号整数且 $x \overset{u}{\leqslant} y$，或是 x、y 都是带符号整数且 $x \leqslant y$（带符号数的比较），那么平均值就是 $(y-x) \overset{u}{\gg} 1$。这样求出来的平均值是向下取整过的，比如 -1 与 0 的平均值是 -1。

2.6 符号扩展

这里所说的“符号扩展”（sign extension），意思是先在字组中确定某个位元为符号位，然后再把它的值向左传播，覆盖掉那些位元中原有的值。要完成此操作，通常的做法是：先进行逻辑左移，再进行带符号的右移。然而若是电脑执行这些指令的速度比较慢，或者根本没有这类指令，那么可以考虑用下列算法之一来计算。此处列出将 7 号位元向左传播所用的三种算式：

$$((x + \mathbf{0x00000080})\ \&\ \mathbf{0x000000FF}) - \mathbf{0x0000\ 0080}$$

$$((x\ \&\ \mathbf{0x000000FF}) \oplus \mathbf{0x00000080}) - \mathbf{0x0000\ 0080}$$

$$(x\ \&\ \mathbf{0x0000\ 007F}) - (x\ \&\ \mathbf{0x00000080})$$

算式中的“+”也可以用“−”或“⊕”代替。第 2 个公式特别有用，因为假如你能确定符号位左方那些即将被替换掉的位元[二]全都是 0，那么其中的按位和操作也可以省了。

2.7 用无符号右移模拟带符号右移操作

如果电脑不支持带符号右移指令，可以考虑使用下列公式来算。第 1 个公式载于[GM]，第 2 个公式根据其原理推算而得。这些公式成立的条件都是 $0 \leqslant n \leqslant 31$，如果电脑有“模 64 移位”[三]指令，那么最后一个公式在 $0 \leqslant n \leqslant 63$ 的情况下均成立。如果认为带符号移位操作调整移位量所用的模除算法与逻辑移位操作相同[四]，那么不管 n 为何值，最后一个式子都成立。

[一] 也就是最左侧的符号位，因为两个带符号的非负数，其符号位肯定都是 0，而一旦两数之和溢出，则在表示运算结果的那个字组中，最左侧的符号位肯定是 1。——译者注

[二] high-order，“高位置的”。一般来说左方的位元权重较高，而且在按照序号称呼字组中的位元时，左侧位元的编号也大，故有此称呼。——译者注

[三] mod-64 shift，该指令在移位前，先要把移位的个数（也就是该操作符的右操作数）调整成一个大于等于 0 且小于 64 的整数，故有此称呼。——译者注

[四] 意思是，如果要实现“模 32 带符号移位”，那么在公式中也要使用“模 32”版本的无符号移位操作，而若要实现“模 64 带符号移位”，则公式中的无符号移位指令也要是“模 64”的才行。只要保证两者相符，那么无论带符号移位操作的移位量是多少，都可以套用这个公式。——译者注

n 若为变量，则实现下述每一条公式需要使用 5 至 6 个基本 RISC 指令。

$$((x+\mathbf{0x8000\ 0000})\overset{u}{\gg}\boldsymbol{n})-(\mathbf{0x8000\ 0000}\overset{u}{\gg}\boldsymbol{n})$$

$$t\leftarrow\overset{u}{\gg}\mathbf{0x8000\ 0000}\overset{u}{\gg}\boldsymbol{n};\qquad((\boldsymbol{x}\overset{u}{\gg}\boldsymbol{n})\oplus\boldsymbol{t})-t$$

$$t\leftarrow(x\ \&\ \mathbf{0x8000\ 0000})\overset{u}{\gg}\boldsymbol{n},(\boldsymbol{x}\overset{u}{\gg}\boldsymbol{n})-(t+t)$$

$$(\boldsymbol{x}\overset{u}{\gg}\boldsymbol{n})\,|\,(-(\boldsymbol{x}\overset{u}{\gg}31)\ll\mathbf{31}-\boldsymbol{n})$$

$$t\leftarrow-(\boldsymbol{x}\overset{u}{\gg}31);\qquad((\boldsymbol{x}\oplus\boldsymbol{t})\overset{u}{\gg}\boldsymbol{n})\oplus\boldsymbol{t}$$

头两个公式中的 $\mathbf{0x8000\ 0000}\overset{u}{\gg}\boldsymbol{n}$ 也可以换成 $\mathbf{1}\ll\mathbf{31}-\boldsymbol{n}$。

如果 n 是常数，那么前两个式子在很多 CPU 中只需 3 个指令就能完成。若 $n=31$，则仅用两个指令即可模拟带符号右移操作：$-(\boldsymbol{x}\overset{u}{\gg}31)$。

2.8 符号函数

下面定义的这个函数称为符号函数（sign function），也称正负号函数（signum function）：

$$\operatorname{sign}(x)=\begin{cases}-1, & x<0,\\ 0, & x=0,\\ 1, & x>0.\end{cases}$$

此函数在大部分电脑中仅需 4 个指令即可实现［Hop］：

$$(\boldsymbol{x}\overset{s}{\gg}31)\,|\,(-\boldsymbol{x}\overset{u}{\gg}31)$$

假若 CPU 没有带符号右移指令，那么可以根据 2.7 节最后所讲的公式来实现。用无符号右移模拟出来的算式非常对称：

$$-(\boldsymbol{x}\overset{u}{\gg}31)\,|\,(-\boldsymbol{x}\overset{u}{\gg}31)$$

如果用了比较谓词的话，3 个指令就够了。以下两种办法都能实现：

$$\begin{aligned}&(\boldsymbol{x}>0)-(\boldsymbol{x}<0),\text{或}\\&(\boldsymbol{x}\geqslant0)-(\boldsymbol{x}\leqslant0)\end{aligned}\tag{4}$$

最后要提一下 $(-\boldsymbol{x}\overset{u}{\gg}31)-(\boldsymbol{x}\overset{u}{\gg}31)$ 这个式子，除了 $\boldsymbol{x}=-\mathbf{2}^{31}$ 的情况外，用它也能正确算出绝对值来。

2.9 三值比较函数

三值比较函数略微扩大了符号函数的使用范围，其定义是：

$$\operatorname{cmp}(x,y)=\begin{cases}-1, & x<y,\\ 0, & x=y,\\ 1, & x>y.\end{cases}$$

该函数有带符号与无符号两个版本，除非明确指出，否则本节所讲内容对二者均适用。通过比较谓词，仅需3个指令即可实现，其计算公式显然是等式（4）的推广：

$$(x>y)-(x<y),\text{或}$$
$$(x\geqslant y)-(x\leqslant y)$$

在Power PC[⊖]上，可以用下列指令［CWG］计算两个无符号数，这种CPU的进位标记会在减法操作未发生借位时置1。[⊜]

```
subf   R5,Ry,Rx     # R5 <-- Rx - Ry.
subfc  R6,Rx,Ry     # R6 <-- Ry - Rx, set carry.
subfe  R7,Ry,Rx     # R7 <-- Rx - Ry + carry, set carry.
subfe  R8,R7,R5     # R8 <-- R5 - R7 + carry, (set carry).
```

要是只能用基本RISC指令的话，恐怕没有什么好办法能计算这个函数。因为像 $x<y$，$x\leqslant y$ 等比较谓词都要大约5条指令才能实现出来（参见2.12节），从而导致整个算法需要大约12个指令（这还是在考虑到可以将那几个重复的 $x<y$ 与 $x>y$ 运算合并的情况下）。如果非要限定基本的RISC指令，那么采用比较与分支的办法也许更好（因为把重复的比较操作合并之后，最差也只要6条指令就能完成）。

2.10 符号传递函数

符号传递函数（transfer of sign function）在Fortran语言中叫做ISIGN，其定义是：

$$\text{ISIGN}(x,y)=\begin{cases}\text{abs}(x), & y\geqslant 0\\ -\text{abs}(x), & y<0\end{cases}$$

在大部分电脑上，只需4个指令就能实现该函数（这样算出来的是函数值与 2^{32} 取模之后的结果）：

$$\begin{aligned}&\boldsymbol{t}\leftarrow \boldsymbol{y}\overset{s}{\gg}31;\\ &\text{ISIGN}(\boldsymbol{x},\boldsymbol{y})=(\text{abs}(\boldsymbol{x})\oplus\boldsymbol{t})-\boldsymbol{t}\\ &\qquad=(\text{abs}(\boldsymbol{x})+\boldsymbol{t})\oplus\boldsymbol{t}\end{aligned}\qquad\qquad \begin{aligned}&\boldsymbol{t}\leftarrow(\boldsymbol{x}\oplus\boldsymbol{y})\overset{s}{\gg}31;\\ &\text{ISIGN}(\boldsymbol{x},\boldsymbol{y})=(\boldsymbol{x}\oplus\boldsymbol{t})-\boldsymbol{t}\\ &\qquad=(\boldsymbol{x}+\boldsymbol{t})\oplus\boldsymbol{t}\end{aligned}$$

2.11 将值为0的位段解码为2的 *n* 次方

有些时候，某个值如果为0或负数是没有意义的，在这种情况下，就可以拿0来表示

⊖ PowerPC（Performance Optimization With Enhanced RISC-Performance Computing，有时简称PPC）是一种精简指令集（RISC）架构的CPU，其基本的设计源自IBM的POWER架构。POWER是1991年由Apple、IBM、Motorola组成的AIM联盟所研发，而PowerPC是AIM联盟平台的一部分。详情参见：http://zh.wikipedia.org/wiki/PowerPC。——译者注

⊜ 原文为“carry”is“not borrow”。在加法操作中，carry flag是表示运算结果是否需要进位的标志，但是在执行减法操作时，计算机却有两种不同的解读方式。有些会在减法需要借位时将此标志置1，而另外一些则相反，在不需借位时置1。PowerPC就属于后者，故作者有此说。详情参见：http://en.wikipedia.org/wiki/Carry_flag。——译者注

2^n，而让其他非零值保持其原意。比如在 PowerPC 的多字节常数加载指令[^1]中，加载长度就是用 5 个位元来表示的。载入长度为 0 的常数是没有意义的，而载入一个 32 字节常数还有些用。此时可以用 0 到 31 来表示待加载常数所占的字节数分别是 1 至 32 个。然而如果换一种方式，"用 0 来表示 32"，那么一旦处理器必须实现那种用寄存器中变量来表示常数长度的指令时（例如 PowerPC 的 `lswx` 指令[^2]），所需的逻辑就会较上一个方案更简单些，因为可以直接使用二进制编码，而不用再加 1 了。

要把一个 1 至 2^n 之间的数编码为"以 0 表示 2^n"的位段，只需要把它和 2^n-1 取掩码就可以了。而解码时如果不想用"测试加分支"的形式来做，就不那么容易了。不过办法还是有的，此处以长度为 3 的位段为例来演示如何解码。若不考虑可能需要的常数加载指令，则实现下列各式均要用 3 条指令。

$$
\begin{array}{lll}
((x-1)\,\&\,7)+1 & ((x+7)\,|\,-8)+9 & 8-(-x\,\&\,7) \\
((x+7)\,\&\,7)+1 & ((x+7)\,|\,8)-7 & -(-x\,|\,-8) \\
((x-1)\,|\,-8)+9 & ((x-1)\,\&\,8)+x &
\end{array}
$$

2.12 比较谓词

"比较谓词"（comparison predicate）是一个用于比较两数大小的函数，它的运算结果只占一个位元：若比较关系**成立**（true），则值为 1；若**不成立**（false），则值为 0。下面列出一些用无分支表达式实现出来的比较谓词，这些式子都会将运算结果放在符号位中。如果想把结果换算成某些语言（例如 C）所用的 1/0 形式，则将运算结果视为无符号数，并右移 31 位[^3]即可，而要将其转换成另外一些语言（如 Basic）所需的 −1/0 形式，那就在执行完代码之后把它带符号右移 31 位。

如果计算机用的 CPU 架构是 MIPS[^4]或书中所描述的 RISC 模型等，那么这些公式就没多大用处了。因为此类指令集都有比较指令，能够直接算出很多比较谓词，并且把结果以 0/1 值的形式放在通用寄存器中。

$$x=y:\qquad \mathrm{abs}(x-y)-1$$

[^1]: load string word immediate，助记符为 `lswi`。详情参考：http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.aixassem/doc/alangref/lswi.htm。——译者注

[^2]: load string word indexed，助记符为 `lswx`，详情参考：http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.aixassem/doc/alangref/lswx.htm。——译者注

[^3]: 原书写为 shift right of 31，但是若 x、y 均为带符号数且比较结果为 1 时，使用 C 语言的右移操作会对其符号进行扩展，而由于运算结果中的"1"处于符号位，所以扩展之后的结果便成了 −1，与文中要求不符。故译文据此略作修改。——译者注

[^4]: MIPS 为 Microprocessor without Interlocked Pipeline Stages 的缩写，是一种采用 RISC 指令集的处理器架构，广泛用在电子产品、网络设备、个人娱乐装置与商业装置上。详情参见：http://zh.wikipedia.org/wiki/MIPS架构。——译者注

$$
\begin{array}{lll}
 & \mathrm{abs}(\boldsymbol{x}-\boldsymbol{y}+\mathbf{0x8000\,000}) & \\
 & \mathrm{nlz}(\boldsymbol{x}-\boldsymbol{y})\ll 26 & \\
 & -(\mathrm{nlz}(\boldsymbol{x}-\boldsymbol{y})\overset{u}{\gg}5) & \\
 & \neg(\boldsymbol{x}-\boldsymbol{y}\mid \boldsymbol{y}-\boldsymbol{x}) & \\
\boldsymbol{x}\neq\boldsymbol{y}: & \mathrm{nabs}(\boldsymbol{x}-\boldsymbol{y}) & \\
 & \mathrm{nlz}(\boldsymbol{x}-\boldsymbol{y})-\mathbf{32} & \\
 & \boldsymbol{x}-\boldsymbol{y}\mid \boldsymbol{y}-\boldsymbol{x} & \\
\boldsymbol{x}<\boldsymbol{y}: & (\boldsymbol{x}-\boldsymbol{y})\oplus[(\boldsymbol{x}\oplus\boldsymbol{y})\,\&\,((\boldsymbol{x}-\boldsymbol{y})\oplus\boldsymbol{x})] & \\
 & (\boldsymbol{x}\,\&\,\neg\boldsymbol{y})\mid((\boldsymbol{x}\equiv\boldsymbol{y})\,\&\,(\boldsymbol{x}-\boldsymbol{y})) & \\
 & \mathrm{nabs}(\mathrm{doz}(\boldsymbol{y},\boldsymbol{x})) & [\mathrm{GSO}] \\
\boldsymbol{x}\leqslant\boldsymbol{y}: & (\boldsymbol{x}\mid\neg\boldsymbol{y})\,\&\,((\boldsymbol{x}\oplus\boldsymbol{y})\mid\neg(\boldsymbol{y}-\boldsymbol{x})) & \\
 & ((\boldsymbol{x}\equiv\boldsymbol{y})\overset{s}{\gg}1)+(\boldsymbol{x}\,\&\,\neg\boldsymbol{y}) & [\mathrm{GSO}] \\
\boldsymbol{x}\overset{u}{<}\boldsymbol{y}: & (\neg\boldsymbol{x}\,\&\,\boldsymbol{y})\mid((\boldsymbol{x}\equiv\boldsymbol{y})\,\&\,(\boldsymbol{x}-\boldsymbol{y})) & \\
 & (\neg\boldsymbol{x}\,\&\,\boldsymbol{y})\mid((\neg\boldsymbol{x}\mid\boldsymbol{y})\,\&\,(\boldsymbol{x}-\boldsymbol{y})) & \\
\boldsymbol{x}\overset{u}{\leqslant}\boldsymbol{y}: & (\neg\boldsymbol{x}\mid\boldsymbol{y})\,\&\,((\boldsymbol{x}\oplus\boldsymbol{y})\mid\neg(\boldsymbol{y}-\boldsymbol{x})) &
\end{array}
$$

如果电脑中有直接能算出绝对值相反数的指令，那上面的公式用起来就方便多了。公式中用“nabs”来表示此函数。和绝对值函数不同，它不会溢出。要是电脑中没有能直接求 nabs 的指令，但有常用的求绝对值（abs）指令，那么就可以用 $-\mathrm{abs}(\boldsymbol{x})$ 来计算 $\mathrm{nabs}(\boldsymbol{x})$。在 $\boldsymbol{x}$ 为负极值时，这么做会溢出两次，但结果却是对的。（此处假设负极值的绝对值是其本身[⊖]）由于有些电脑既没有 abs 指令，也没有 nabs 指令，所以书中还给出了不需要此类指令的算法。

“nlz”函数的值是其自变量中前导 0 的个数。“doz”函数（“求差或取零”函数，difference or zero）的定义在第 46 页。想要算出 $\boldsymbol{x}>\boldsymbol{y}$ 和 $\boldsymbol{x}\geqslant\boldsymbol{y}$ 等比较谓词的值，只需将 $\boldsymbol{x}<\boldsymbol{y}$ 与 $\boldsymbol{x}\leqslant\boldsymbol{y}$ 等公式中的 $\boldsymbol{x}$、$\boldsymbol{y}$ 互换即可。公式中凡出现 $\boldsymbol{x}$、$\boldsymbol{y}$、$\boldsymbol{x}-\boldsymbol{y}$ 等值与 **0x8000 0000** 的加法操作，均可用任意能够反转最高位的指令代替。

由于比较谓词 $\boldsymbol{x}<\boldsymbol{y}$ 的值与 $\boldsymbol{x}/2-\boldsymbol{y}/2$ 的符号位相符，而且这个减法算式又不会溢出，所以可由此论述推导出另外一组公式。为了防止移位操作把关键信息丢掉，还需酌情对运算结果减 1 以修正其值。此算法公式如下：

$$
\begin{array}{ll}
\boldsymbol{x}<\boldsymbol{y}: & (\boldsymbol{x}\overset{s}{\gg}1)-(\boldsymbol{y}\overset{s}{\gg}1)-(\neg\boldsymbol{x}\,\&\,\boldsymbol{y}\,\&\,\mathbf{1}) \\
\boldsymbol{x}\overset{u}{<}\boldsymbol{y}: & (\boldsymbol{x}\overset{u}{\gg}1)-(\boldsymbol{y}\overset{u}{\gg}1)-(\neg\boldsymbol{x}\,\&\,\boldsymbol{y}\,\&\,\mathbf{1})
\end{array}
$$

上面这两种算法需要 7 个指令（如果按位与和按位取反能用 1 个指令来完成，那就是

⊖ 假如字组的位宽是 32，那么负极值就是 -2^{31}，而书中介绍的 $-\mathrm{abs}$ 算法若要成立，必须假设 -2^{31} 的绝对值仍然是 -2^{31}。——译者注

6 个)，和刚才讲的那组公式比起来并不算好（刚才那组公式所需的指令数，会依所用指令集中逻辑指令的丰富程度而变，范围在 5 到 7 个之间）。

上述公式中用到了 nlz 函数的那个算法是由［Shep］给出的，他计算 $x=y$ 这个谓词时的算式特别有用，因为只要稍加修改，就可以变成下面这个样子。新算式只用 3 条指令就可以把比较谓词的值化成 1/0 形式：

$$\mathrm{nlz}(x-y) \overset{u}{\gg} 5$$

带符号数与 0 的比较也很常用，所以这里有必要特别讲一下。下面这些公式都可以计算比较谓词，它们差不多都是从上面那一套公式直接推出来的，而且运算结果也是放在符号位里面。

$$
\begin{array}{lll}
x=0: & \mathrm{abs}(x)-1 & \\
 & \mathrm{abs}(x+\mathbf{0x8000\ 0000}) & \\
 & \mathrm{nlz}(x) \ll 26 & \\
 & -(\mathrm{nlz}(x) \overset{u}{\gg} 5) & \\
 & \neg(x \mid -x) & \\
 & \neg x \;\&\; (x-1) & \\
x \neq 0: & \mathrm{nabs}(x) & \\
 & \mathrm{nlz}(x)-32 & \\
 & x \mid -x & \\
 & (x \overset{u}{\gg} 1)-x & \text{[CWG]} \\
x<0: & x & \\
x \leqslant 0: & x \mid (x-1) & \\
 & x \mid \neg -x & \\
x>0: & x \oplus \mathrm{nabs}(x) & \\
 & (x \overset{s}{\gg} 1)-x & \\
 & -x \;\&\; \neg x & \\
x \geqslant 0: & \neg x &
\end{array}
$$

把无符号比较谓词中的操作数向上偏移 2^{31}，就得到了带符号版本的比较谓词，其逆向变换依然成立[㊀]。所以可得如下二式：

$$x<y = x+2^{31} \overset{u}{<} y+2^{31}$$

$$x \overset{u}{<} y = x-2^{31} < y-2^{31}$$

对于判断 $\leqslant$ 和 $\overset{u}{\leqslant}$ 关系的比较谓词来说，以上关系式仍然适用。这些算式中 2^{31} 前面的

㊀ 在 Java 语言中比较无符号数时可用此式，因为 Java 语言没有无符号型整数。

运算符，不论是加号、减号，还是按位异或，结果都一样，因为这么做只是为了反转符号位而已。如果 CPU 支持基本 RISC 指令集中的“带移位常量加法”指令（add immediate shifted），那么就不用再多花两条指令把 2^{31} 这个常数加载到寄存器中了。

当 x 与 y 正负号相同时，$x<y=x\overset{u}{<}y$，而两者正负号不同时，$x<y=x\overset{u}{>}y$ [Lamp]，基于这一原理，可以推出另一种用无符号比较谓词来比较带符号数的方法。与刚才一样，逆向变换也成立。于是可以得出两个公式：

$$x<y=(x\overset{u}{<}y)\oplus x_{31}\oplus y_{31}$$

$$x\overset{u}{<}y=(x<y)\oplus x_{31}\oplus y_{31}$$

其中 x_{31} 和 y_{31} 分别表示 x 与 y 的符号位。同样的关系式也适用于$\leqslant$和$\overset{u}{\leqslant}$等比较谓词。

任选一个除了$=$和$\neq$之外的比较谓词，都可以根据上述各种关系式用其他形式的比较谓词把它表示出来，而且在大多数电脑上最多只需 3 个指令。因为 $x\overset{u}{\leqslant}y$ 这个比较谓词实现起来很容易（它的值也就是执行 $y-x$ 之后的进位标志），所以我们就以它为例来推算其他比较谓词：

$$x<y=\neg(y+2^{31}\overset{u}{\leqslant}x+2^{31})$$

$$x\leqslant y=x+2^{31}\overset{u}{\leqslant}y+2^{31}$$

$$x>y=\neg(x+2^{31}\overset{u}{\leqslant}y+2^{31})$$

$$x\geqslant y=y+2^{31}\overset{u}{\leqslant}x+2^{31}$$

$$x\overset{u}{<}y=\neg(y\overset{u}{\leqslant}x)$$

$$x\overset{u}{>}y=\neg(x\overset{u}{\leqslant}y)$$

$$x\overset{u}{\geqslant}y=y\overset{u}{\leqslant}x$$

2.12.1 利用进位标志求比较谓词

若是计算机很容易就能把进位标志的值放到一个通用寄存器里，那么可以用该标志准确判断出一些比较谓词的值。下面列出一些此类关系式。其中 carry（expression）这个写法表示 expression 中最外层运算的标志位。此处假定进位标志是加法器用 $x+\bar{y}+1$ 的方式来计算减法操作 $x-y$ 时产生的，其值与减法的“借位”情况相反[⊖]。

$x=y$：　carry$(\mathbf{0}-(x-y))$, or carry$((x+\bar{y})+\mathbf{1})$, or
carry$((x-y-\mathbf{1})+\mathbf{1})$

⊖ 原文为 the complement of “borrow”，意思是，如果减法操作没有发生借位，则进位标志是 1，若发生借位，则该标志位为 0。——译者注

$x \neq y$:　carry$((x-y)-1)$, i. e. , carry$((x-y)+(-1))$

$x < y$:　$\neg$carry$((x+2^{31})-(y+2^{31}))$, or $\neg$carry$(x-y) \oplus x_{31} \oplus y_{31}$

$x \leqslant y$:　carry$((y+2^{31})-(x+2^{31}))$, or carry$(y-x) \oplus x_{31} \oplus y_{31}$

$x \overset{u}{<} y$:　$\neg$carry$(x-y)$

$x \overset{u}{\leqslant} y$:　carry$(y-x)$

$x = 0$:　carry$(0-x)$, or carry$(\bar{x}+1)$

$x \neq 0$:　carry$(x-1)$, i. e. , carry$(x+(-1))$

$x < 0$:　carry$(x+x)$

$x \leqslant 0$:　carry$(2^{31}-(x+2^{31}))$

若想求 $x>y$ 的值，只需把 $x \leqslant y$ 的结果取反就好，判断其他包含“大于”的关系式是否成立时也是如此：对与之相反的比较谓词求补[1]即可。

在 IBM RS/6000 计算机和其兄弟机型 PowerPC 系列电脑上，都采用 GNU Superoptimizer[2]来求谓词表达式的值［GK］。RS/6000 电脑不仅具备计算 abs(x)、nabs(x)、doz(x,y) 函数的指令，而且还有各种形式的加减法指令，这些指令都能把进位标志考虑在内。有人发现，RS/6000 最多用 3 条基本指令（也就是只花 1 个 CPU 周期就能执行完的指令）就能算出每一种整数谓词表达式的值，这个结果连此计算机的设计师都觉得吃惊。这里所说的“每一种”，包括 6 种判断两个带符号数的比较谓词，以及 4 种判断两个无符号数的比较谓词。上述比较谓词的第 2 个操作数都是 0，而且结果均是 1/0 或 −1/的形式。在缺少 abs(x)、nabs(x) 及 doz(x,y) 的 PowerPC 中，实现上述各种谓词表达式最多需要 4 条指令。

2.12.2 计算机如何设置比较谓词

大多数电脑都会用某种方式将整数比较谓词的值化为 1 个位元。有些 CPU 把表示比较结果的位元存放在“条件寄存器”（condition register）中，而另外一些（比如本书用到的 RISC 模型）则把它放在通用寄存器里。不管采用那种实现，都需要有一套机制把比较谓词的两个操作数相减，并对包含运算结果的那些二进制位做几个逻辑操作，以提取出一个表示比较结果的位元。

下面列出了求各种比较谓词所用的逻辑操作。假定电脑用 $x+\bar{y}+1$ 来计算 $x-y$，而且运算完毕后可从结果中得知这几个量的值：

C_0：计算最高位时产生的进位值。

---

[1] 由于比较谓词的值只有一个位元，非 0 即 1，所以这里“求补”的意思就是把 1 变为 0，把 0 变为 1。——译者注

[2] 是一个能够实现超级优化技术（Superoptimization）的程序，详情参见：http：//en. wikipedia. org/wiki/Superoptimization 及 http：//ftp. gnu. org/gnu/superopt/。——译者注

C_i：计算次高位时，带给最高位的进位值。

N：运算结果的符号位。

Z：若运算结果与C_0求异或后，每一位都是0，则该值为1，否则为0。

有了上述各量，就可以用如下布尔代数操作来求比较谓词了（两值毗邻表示求其逻辑与，而+代表逻辑或）：

V：　$C_i \oplus C_o$(两个带符号数相减是否溢出)

$x=y$：　Z

$x \neq y$：　$\bar{Z}$

$x<y$：　$N \oplus V$

$x \leqslant y$：　$(N \oplus V)+Z$

$x>y$：　$(N \equiv V)\bar{Z}$

$x \geqslant y$：　$N \equiv V$

$x \overset{u}{<} y$：　$\overline{C_o}$

$x \overset{u}{\leqslant} y$：　$\overline{C_o}+Z$

$x \overset{u}{>} y$：　$C_o\bar{Z}$

$x \overset{u}{\geqslant} y$：　C_o

2.13 溢出检测

“溢出”是指算术操作的结果太大或太小，以致无法将其正确表示到目标寄存器里。本节讨论一些程序员可能会用到的溢出检测法，它们都无需使用CPU中现成的“状态位”(Status Bit)。这么做意义很大，因为有些CPU（例如MIPS架构的）没有这种状态位，而且即便有，某些高级语言也很难或根本无法访问它们。

2.13.1 带符号的加减法

当今的CPU在计算整数加减法时如果发生溢出，则总是会丢弃运算结果的最高位，并且把加法器中较低的那些位元照原样存起来。当且仅当两操作数符号相同但二者之和却与其符号不同时，带符号整数加法才会溢出。让人觉得奇怪的是：哪怕加法器在运算时把进位也带进来，其结果仍符合上述规律。也就是说，在计算 $\boldsymbol{x}+\boldsymbol{y}+1$ 时，只要 $\boldsymbol{x}$、$\boldsymbol{y}$ 的符号不同，便怎么也溢出不了。这一规律对计算多字带符号数的加法很重要，因为此种加法在最后一步需要把两个字组与从低位带过来的进位一并相加，而这个进位可能是0，也有可能是+1。

要证明上述加法规律，现在假定两个被加数 x、y 均为一个字组长的带符号整数，而

c（执行运算前的进位标志[⊖]）是 0 或 1，为简化证明过程，规定字组位宽是 4。若 x 与 y 符号不同，则：

$$-8 \leqslant x \leqslant -1，且$$
$$0 \leqslant y \leqslant 7$$

若 x 为非负数而 y 为负数，则其取值范围与之类似。不管哪种情况，我们都把两个不等式相加，并且将 c 的值可取 0 或 1 这一点考虑在内[⊜]，于是得出：

$$-8 \leqslant x+y+c \leqslant 7$$

4 位带符号数显然能够容纳上述取值范围，因此两操作数符号不同时是不会溢出的。

若 x 与 y 符号相同，则分两种情况：

$$\begin{array}{cc} \text{(a)} & \text{(b)} \\ -8 \leqslant x \leqslant -1 & 0 \leqslant x \leqslant 7 \\ -8 \leqslant y \leqslant -1 & 0 \leqslant y \leqslant 7 \end{array}$$

在上述两种情况下，可分别推出：

$$\begin{array}{cc} \text{(a)} & \text{(b)} \\ -16 \leqslant x+y+c \leqslant 1 & 0 \leqslant x+y+c \leqslant 15 \end{array}$$

若求和结果在下述取值范围内，则无法表示为 4 位带符号整数，于是就会溢出：

$$\begin{array}{cc} \text{(a)} & \text{(b)} \\ -16 \leqslant x+y+c \leqslant -9 & 8 \leqslant x+y+c \leqslant 15 \end{array}$$

式（a）所述的情况，相当于在容纳求和结果的 4 个位元中，最高位是 0，而这与 x、y 的符号相反。式（b）所述的情况，则相当于在容纳求和结果的 4 个位元中，最高位是 1，这也和 x、y 的符号相反。

在多字整数的减法操作中，需要计算 $\boldsymbol{x-y-c}$，此处 $\boldsymbol{c}$ 的值还是 **0** 或 **1**，其中 **1** 表示由低位字组产生的借位（borrow-in）。采用与上述分析过程类似的方式，也可证明：当且仅当 $\boldsymbol{x}$ 与 $\boldsymbol{y}$ 符号相反，而 $\boldsymbol{x-y-c}$ 的符号又与 $\boldsymbol{x}$ 相反（也可以说成“与 y 相同”）时，$\boldsymbol{x-y-c}$ 才会溢出。

基于此规律，可以用以下表达式判别溢出谓词，运算结果存放在符号位中。如果套用公式后再右移或带符号右移 31 位，那么就可以将其化为 1/0 或 −1/0 形式的值。

$$\begin{array}{ll} \boldsymbol{x+y+c} & \boldsymbol{x-y-c} \\ (\boldsymbol{x} \equiv \boldsymbol{y}) \,\&\, ((\boldsymbol{x+y+c}) \oplus \boldsymbol{x}) & (\boldsymbol{x} \oplus \boldsymbol{y}) \,\&\, ((\boldsymbol{x-y-c}) \oplus \boldsymbol{x}) \\ ((\boldsymbol{x+y+c}) \oplus \boldsymbol{x}) \,\&\, ((\boldsymbol{x+y+c}) \oplus \boldsymbol{y}) & ((\boldsymbol{x-y-c}) \oplus \boldsymbol{x}) \,\&\, ((\boldsymbol{x-y-c}) \equiv \boldsymbol{y}) \end{array}$$

如果用第 1 列的第 2 种形式或第 2 列的第 1 种形式来判断溢出（也就是不含按位等值操作≡的那两个公式），那么在本书所定义的基本 RISC 指令集范围内，除去计算 $\boldsymbol{x+y+c}$

⊖ 本书中用 carry-in 表示这个概念，如果执行的是带进位的加法（add with carry），则在求和时还要将该值一并加入。——译者注

⊜ 也就是将 $-8 \leqslant x \leqslant -1$、$0 \leqslant y \leqslant 7$、$0 \leqslant c \leqslant 1$ 这三个不等式加起来。——译者注

或 $x-y-c$ 所需的指令外，仅需3条即可。若想用分支代码来处理溢出，那么再加一条负分支（branch if negative）指令即可。

执行代码时若启用了溢出中断，则程序员可能想在不引发溢出的情况下预判某个加减法操作是否会溢出。以下列出无分支的实现方式：

$x+y+c$	$x-y-c$
$z \leftarrow (x \equiv y) \,\&\, \mathtt{0x8000\ 0000}$	$z \leftarrow (x \oplus y) \,\&\, \mathtt{0x8000\ 0000}$
$z \,\&\, (((x \oplus z)+y+c) \equiv y)$	$z \,\&\, (((x \oplus z)-y-c) \oplus y)$

在左边一栏中，如果 x 与 y 符号相同，则 $z=\mathtt{0x8000\ 0000}$，若符号不同，则 $z=0$。在第2个表达式的加法操作中，$x\oplus z$ 与 y 的符号相反，所以肯定不会溢出。若 x 与 y 都是非负数，则当且仅当（$x-2^{31}$）$+y+c\geqslant 0$ 时，也就是 $x+y+c\geqslant 2^{31}$ 时，第2个表达式的值才会是1。而其中的 $x+y+c\geqslant 2^{31}$ 也就是 $x+y+c$ 发生溢出的条件。若 x 与 y 都是负数，则当且仅当（$x+2^{31}$）$+y+c<0$ 时，也就是 $x+y+c<-2^{31}$ 时，第2个表达式的值才会是1。而其中的 $x+y+c<-2^{31}$ 也就是此时能发生溢出的条件。为了保证在 x 与 y 符号不同时能够得到正确结果（也就是让符号位的值是0），还要与 z 按位取和才行。对于减法操作的溢出判断（也就是右侧那一栏）也可以按相似的思路分析。实现上述代码需要9条RISC指令。

要是可以使用加法操作的进位标志，那它也许能用来计算带符号数的溢出谓词。虽说未必真的如此，但是通过这一思路可以构想出下面这种算法。

如果 x 为带符号整数，则 $x+2^{31}$ 可以用无符号整数表示出来，方法就是把 x 的最高位反转。若 $x+y\geqslant 2^{31}$，也就是 $(x+2^{31})+(y+2^{31})\geqslant 3\cdot 2^{31}$ 时，两数之和会发生正向（positive direction）溢出。而溢出条件中的后一种写法，实际上就是说，在无符号数加法中，进位标志（两数之和大于等于 2^{32}）与求和结果的最高位都是1。同理，当进位标志与求和结果的最高位均是0时，会发生负向溢出。

于是可以得出一种检测带符号数加法是否溢出的算法：

计算 $(x\oplus 2^{31})+(y\oplus 2^{31})$，设求和结果为 s，进位标志为 c。

当且仅当 c 与 s 的最高位相同时，才会发生溢出。

对于两个无符号数，这么算出来的求和结果是正确的，因为同时反转两个操作数的最高位，并不会改变它们的和。

判断减法是否溢出也可以用此算法，只是要把第1步中的加号换成减号。此时假定计算机通过 $x+\bar{y}+1$ 来计算 $x-y$ 的值，并且进位标志的值也是在计算 $x+\bar{y}+1$ 的过程中产生的。对于两个带符号数的减法来说，这种算法计算出来的差也是正确的。

这几个公式也许很有意思，不过在大部分电脑上，它们的效率都低于那些根本不用进位标志的式子（比如用 $(x\equiv y)\,\&\,(s\oplus x)$ 判断加法是否溢出，用 $(x\oplus y)\,\&\,(d\oplus x)$ 来判断减法是否溢出，其中 s、d 分别表示 x、y 的和与差）。

2.13.2 计算机执行带符号数的加减法时如何设置溢出标志

大部分 CPU 都会根据“次高位向最高位带入的进位是否与最高位带出的进位不同”这一逻辑来判断是否需要设置“溢出”标志。说来也怪，如果假定 $x-y$ 是通过 $x+\bar{y}+1$ 算出来的，那么这条标准还真能准确判断出加减法是否会溢出。而且，无论是做带进位的加减法还是不带进位的加减法，这条规律都成立。若要在软件中计算带符号数的溢出谓词，那么用这条规律恐怕推算不出什么好办法，虽说如此，不过它还是带来了一种简便的方式，可用于计算从次高位向最高位带入的进位。对于加法减法操作来说，运算完下列表达式之后（其中 c 为 $\mathbf{0}$ 或 $\mathbf{1}$），符号位中的值就是在运算过程中由次高位带入最高位中的进位/借位：

进位	借位
$(x+y+c) \oplus x \oplus y$	$(x-y-c) \oplus x \oplus y$

其实在该表达式的运算结果中，序号为 i 的位元值就是在计算过程中由其右侧位元所带入位置 i 的进位或借位。

2.13.3 无符号数的加减法

下列无分支代码可以计算无符号数加减法的溢出谓词，运算结果放在符号位中。带右移的那个表达式只有在 $c=\mathbf{0}$ 时才值得用[⊖]。表达式里中括号之内的部分，用于计算从最低有效位中生成的进位或借位。

$$x+y+c\text{，无符号数的加法}$$

$$(x \mathbin{\&} y) \mid ((x \mid y) \mathbin{\&} \neg(x+y+c))$$

$$(x \overset{u}{\gg} 1)+(y \overset{u}{\gg} 1)+[((x \mathbin{\&} y) \mid ((x \mid y) \mathbin{\&} c)) \mathbin{\&} 1]$$

$$x-y-c\text{，无符号数的减法}$$

$$(\neg x \mathbin{\&} y) \mid ((x \equiv y) \mathbin{\&} (x-y-c))$$

$$(\neg x \mathbin{\&} y) \mid ((\neg x \mid y) \mathbin{\&} (x-y-c))$$

$$(x \overset{u}{\gg} 1)-(y \overset{u}{\gg} 1)-[((\neg x \mathbin{\&} y) \mid ((\neg x \mid y) \mathbin{\&} c)) \mathbin{\&} 1]$$

通过比较操作，可以用极为简单的公式判断出无符号加减法是否溢出 [MIPS]。对于无符号加法来说，只要二数之和小于（无符号比较）其中任何一个操作数，那就表明溢出了（同时也意味着最高位会带出进位）。这个算法与另外几个和它相似的公式都列在下面。可惜的是，没办法把进位标志或借位标志以变量 c 的形式纳入到公式中。这样的话，程序在运算前必须先测试 c，并根据其值是 $\mathbf{0}$ 还是 $\mathbf{1}$ 来套用不同类型的比较公式。

⊖ 因为若 c 不为 0，则不带右移的算法比它所需的指令数少。——译者注

$x+y$，无符号数加法	$x+y+1$，无符号数加法	$x-y$，无符号数加法	$x-y-1$，无符号数加法
$\neg x \overset{u}{<} y$	$\neg x \overset{u}{\leqslant} y$	$x \overset{u}{<} y$	$x \overset{u}{\leqslant} y$
$x+y \overset{u}{<} x$	$x+y+1 \overset{u}{\leqslant} x$	$x-y \overset{u}{>} x$	$x-y-1 \overset{u}{\geqslant} x$

每种情况下的第 1 个公式，都可以在执行加减法前先判断出是否溢出，这就提供了一个在不引发溢出情况下的检测办法。而第 2 个公式则必须先执行完可能产生溢出的加减法操作，然后才能得知结果。

似乎没有一套与之相似的简单公式，让我们能通过比较操作求出带符号数的溢出谓词。

2.13.4 乘法

对于乘法来说，溢出就意味着操作结果无法用 32 个位元表达出来（不管是带符号数乘法，还是无符号数乘法，其乘积总可以用 64 个位元来表示）。若是能获取乘积的高 32 位，那么溢出检测就很容易了。在表示乘积的 64 个位元中，设其左右两端的 32 个位元分别为 $\text{hi}(x\times y)$ 与 $\text{lo}(x\times y)$，则可用下列公式求溢出谓词［MIPS］：

$x\times y$，无符号数乘法	$x\times y$，带符号数乘法
$\text{hi}(x\times y)\neq 0$	$\text{hi}(x\times y)\neq(\text{lo}(x\times y)\overset{s}{\gg}31)$

还有一种检测乘法溢出的方法，是通过对乘积做除法而得出的。使用这种方法时需注意，不要让 0 做除数，而且用此方法判断带符号乘法是否溢出显得更为复杂。若下列表达式为真，则表明发生溢出了：

带符号数	无符号数
$z\leftarrow x*y$	$z\leftarrow x*y$
$y\neq 0\ \vec{\&}\ z\overset{u}{\div}y\neq x$	$(y<0\ \&\ x=-2^{31})\mid(y\neq 0\ \vec{\&}\ z\div y\neq x)$

复杂之处就在于，如果 $x=-2^{31}$ 而 $y=-1$，那么此时乘法操作将溢出。而 CPU 却很可能会认为此操作的结果是 -2^{31}，这样一来，除法操作也溢出了，其结果（在某些电脑上）可能是任意值。因此，必须单独检测这种情形，公式中的 $y<0\ \&\ x=-2^{31}$ 就是用来执行此判断的。上述表达式还用“条件与”（conditional and）操作符来预防 0 做除数的情况（在 C 语言中可用&&操作符）。

不执行乘法操作（即在不引发溢出的情况下）也可以判定是否会溢出。对无符号整数来说，当且仅当 $xy>2^{32}-1$，也就是 $x>((2^{32}-1)/y)$ 时，乘积才会溢出。由于 x 为整数，故可将条件改写为 $x>\lfloor(2^{32}-1)/y\rfloor$。用计算机算术表示，就是：

$$y\neq 0\ \vec{\&}\ x\overset{u}{>}(\mathbf{0xFFFFFFFF}\overset{u}{\div}y)$$

对于带符号数来说，想判断 $x*y$ 是否溢出就不那么简单了。假如 x 与 y 符号相同，那么当且仅当 $xy>2^{31}-1$ 时，乘积才溢出，而如果 x 与 y 符号不同，则当且仅当 $xy<-2^{31}$ 时，乘积才会溢出。可以按照表 2.2 中列出的不同情况来判断，该表用到了带符号数的除法操作。因为要分 4 种情况，所以这种测试很难实现。正是由于除法溢出以及无法

用带符号数表示 $+2^{31}$ 这两个问题，才导致表中的 4 个表达式不好统一。

表 2.2　检测带符号乘法是否溢出所用的表达式

	$y>0$	$y\leqslant 0$
$x>0$	$x>\mathrm{0x7FFFFFFF}\div y$	$y<\mathrm{0x8000\ 0000}\div x$
$x\leqslant 0$	$x<\mathrm{0x8000\ 0000}\div y$	$x\neq 0\ \vec{\&}\ y<\mathrm{0x7FFFFFFF}\div x$

要是能用无符号除法的话，那么测试过程就可以简化了。x 与 y 的绝对值是可以用无符号数正确表示出来的。由此，我们可以把 4 种情况一起合并为下述算法。如果 x 与 y 符号相同，则变量 $c=2^{31}-1$，否则 $c=2^{31}$。

$$
\begin{aligned}
&c \leftarrow ((x \equiv y) \overset{s}{\gg} 31) + 2^{31}\\
&x \leftarrow \mathrm{abs}(x)\\
&y \leftarrow \mathrm{abs}(y)\\
&y \neq 0\ \vec{\&}\ x \overset{u}{>} (c \overset{u}{\div} y)
\end{aligned}
$$

前导 0 计数指令（the number of leading zero instruction）可用来估算 $x*y$ 是否溢出，并且可调整估算结果，以准确判断溢出情况。首先来看无符号数乘法。如果 x 与 y 都是 32 位元的无符号量，并且分别具有 m 及 n 个前导 0，那么很明显，在表示两者乘积的 64 位元中，前导 0 的个数要么是 $m+n$，要么是 $m+n+1$（如果 $x=0$ 或 $y=0$，则前导 0 的个数为 64）。如果 64 位乘积的前导 0 个数小于 32，则表明发生溢出了。因此：

$\mathrm{nlz}(x)+\mathrm{nlz}(y)\geqslant 32$：乘法运算肯定不会溢出。

$\mathrm{nlz}(x)+\mathrm{nlz}(y)\leqslant 30$：乘法运算必定溢出。

如果 $\mathrm{nlz}(x)+\mathrm{nlz}(y)=31$，那么有时会溢出，有时则不会。在这种情况下，可以用 $t=x\lfloor y/2\rfloor$ 来判断。计算 t 时不会溢出。如果 y 是偶数，xy 就是 $2t$，若是奇数，就是 $2t+x$，所以当 $t\geqslant 2^{31}$ 时，xy 的乘积就会溢出。上述思路可以归结为一种计算 xy 的方法，并且让程序在发生溢出时转入名为“overflow”的分支。该方案的代码如图 2.2 所示。

```
unsigned x, y, z, m, n, t;

m = nlz(x);
n = nlz(y);
if (m + n <= 30) goto overflow;
t = x*(y >> 1);
if ((int)t < 0) goto overflow;
z = t*2;
if (y & 1) {
   z = z + x;
   if (z < x) goto overflow;
}
// z is the correct product of x and y.
```

图 2.2　判断无符号乘法是否溢出

对于带符号数的乘法，我们可以在操作数为非负时计算前导 0 的个数，而在操作数为负数时计算前导 1 的个数，并由此推算出部分结果。设

$$
\begin{aligned}
&m=\mathrm{nlz}(x)+\mathrm{nlz}(\bar{x}),\text{且}\\
&n=\mathrm{nlz}(y)+\mathrm{nlz}(\bar{y})
\end{aligned}
$$

则可得出如下两条论断：

$m+n\geqslant 34$：乘法操作绝对不会溢出。

$m+n \leqslant 31$：乘法操作必定溢出。

$m+n$ 为 32 或 33 时需要具体分析。如果 $m+n=33$，那么只有当参与乘法的两数均为负，且正确的乘积恰好是 2^{31}（这个值无法用 32 位带符号数正确表示出来，会被计算机当成 -2^{31}）时，才表明发生了溢出。所以，可通过判断乘积的符号是否正确来得知溢出情况（也就是说，如果 $x \oplus y \oplus (x*y) < 0$，则乘法溢出）。若 $m+n=32$，则不太容易判断是否溢出。

此处不打算深究这个问题了，只是想再补充一句：带符号乘法的溢出还可以用 $\text{nlz}(\text{abs}(x))+\text{nlz}(\text{abs}(y))$ 判断，然而和上面一样，这个式子也有两种尚需讨论的情况（分别是当求和结果为 31 或 32 时）。

2.13.5 除法

对于带符号数除法 $x \div y$ 来说，若此表达式结果为真，则发生溢出：

$$y=0 \mid (x=\text{0x8000 0000} \;\&\; y=-1)$$

大部分计算机在遇到 $0 \div 0$ 这种结果为定义的情况时，都会产生溢出（或中断）信号。

如果算上最后处理溢出的分支，那么直接用代码来计算这个表达式需要 7 条指令，其中 3 条是分支指令。似乎没有特别好的办法能改进这个算法，不过可以试试下面这种方案：

$$[\text{abs}(y \oplus \text{0x8000 0000}) \mid (\text{abs}(x) \;\&\; \text{abs}(y \equiv \text{0x8000 0000}))] < 0$$

上面这个式子意思是说，先计算中括号内的大表达式，如果结果小于 0，就转入分支。若是计算机支持算式中用到的指令，并且能够迅速“和 0 比较”，那么算上常数加载和最后的分支，大约需要 9 条指令。

还有一种办法，先用下式算出一个 z 值（在很多电脑中需要 3 条指令）：

$$z \leftarrow (x \oplus \text{0x8000 0000}) \mid (y+1)$$

然后若发现 $y=0 \mid z=0$，就直接转入分支，否则用下面任意一个式子检测溢出：

$$((y \mid -y) \;\&\; (z \mid -z)) \geqslant 0$$

$$(\text{nabs}(y) \;\&\; \text{nabs}(z)) \geqslant 0$$

$$((\text{nlz}(y) \mid \text{nlz}(z)) \overset{u}{\gg} 5) \neq 0$$

如果电脑支持上面用到的那些指令，则上述三种算法所需的总指令数分别为 9 个，7 个，8 个。最后一个式子很适合在 PowerPC 上使用。

对于无符号除法 $x \overset{u}{\div} y$ 来说，当且仅当 $y=0$ 时，才会溢出。

在某些有“长除法”指令（“long division” instruction，参见 9.4 节）的电脑上，我们可能想要用基本指令来预测此操作会不会溢出。在讨论这个问题时，假定该指令会用双字（doubleword）除以单字[⊖]（fullword），其商也是单字，此外有可能还会产生一个单

⊖ 原文为 fullword，直译是“全字”，此处就指一个字组，只是为了和双字（doubleword）对称，故加了“full”。——译者注

字的余数。

若除数为 0 或商无法以 32 位表示，则此类除法指令就会溢出。一旦溢出了，那么商和余数通常都是错的。余数不会因为其值太大、无法用 32 个位元表达出来而溢出（因为它从值上看肯定比除数小），所以测试除数是否正确的方式与测试商所用的方法一样。

我们假定电脑有 64 位或 32 位的通用寄存器，并且能够对 64 位的值进行基本操作（诸如移位、加法等）。比方说编译器或许会实现一种双字节整数类型。

无符号的情况比较容易判断：设 x 为双字，y 为单字，当（且仅当）满足下列任意一个条件时，$x \div y$ 的商才会溢出。这两个判别式是等效的。

$$y \neq 0 \ \& \ x < (y \ll 32)$$

$$y \neq 0 \ \& \ (x \overset{u}{\gg} 32) < y$$

在 32 位机上不需要移位，只比较 y 和存放 x 高 32 位的寄存器即可，而要保证 64 位机结果正确，则必须检查除数 y 是否为一个 32 位的量（例如，判断（$y \overset{u}{\gg} 32$）$=0$）。

带符号的情况就更有意思了。首先必须保证 $y \neq 0$，而且如果是 64 位机，还要核实 y 的值能够用 32 个位元正确表示出来（检查（$(y \ll 32) \overset{s}{\gg} 32 = y$ 是否成立）。假设上述条件都满足，那么根据被除数与除数的正负性，可按照下表列出的 4 种情况，准确判断出商是否能够容纳在 32 位元的字组中。表格中的表达式用的是普通算术，而非计算机算术。

每一栏中的关系式都由其上方的关系式推导而来，这些关系式之间互为充要条件。利用 9.1 节定理 D1 中的某些关系式，可以去掉表格中的向下取整函数与向上取整函数。

$x \geq 0, y > 0$	$x \geq 0, y < 0$	$x < 0, y > 0$	$x < 0, y < 0$
$\lfloor x/y \rfloor < 2^{31}$	$\lceil x/y \rceil \geq -2^{31}$	$\lceil x/y \rceil \geq -2^{31}$	$\lfloor x/y \rfloor < 2^{31}$
$x/y = 2^{31}$	$\lceil x/y \rceil > -2^{31} - 1$	$\lceil x/y \rceil > -2^{31} - 1$	$x/y < 2^{31}$
$x < 2^{31} y$	$x/y > -2^{31} - 1$	$x/y > -2^{31} - 1$	$x > 2^{31} y$
	$x < -2^{31} y - y$	$x > -2^{31} y - y$	$-x < 2^{31}(-y)$
	$x < 2^{31}(-y) + (-y)$	$-x < 2^{31} y + y$	

现在演示一下此表的用法。以最左边一栏为例，它适用于 $x \geq 0$ 且 $y > 0$ 的情况，此时商为 $\lfloor x/y \rfloor$。要想把它表示为 32 位元的量，必须保证此值小于 2^{31}。由此可推出：实数 x/y 必须小于 2^{31}，或者说 x 必须小于 $2^{31} y$。要判断这一关系式，可以把 y 左移 31 位，并与 x 相比。

当 x 与 y 的符号不同时，商一般写为 $\lceil x/y \rceil$，由于商为负数，所以其最小值是 -2^{31}。

每一栏中最后一行的比较表达式用的都是同一种比较关系（也就是小于）。鉴于 x 可能会取负极值，所以 3、4 两列中的表达式必须使用无符号比较操作。而前两栏中那些待比较的数值，其最高位均为 0，所以此处也可以执行无符号比较。

这 4 种测试当然可以用条件分支统一起来：只需按情况执行不同的判断算法，然后根据最终比较结果转入处理溢出或不溢出的分支即可。然而我们观察到一个现象：当 y 为负值时，相应的判别式与 y 为正值时的式子相差无几，x 也是这样。因此，可以利用这一

规律减少分支数量，通过使用 x 与 y 的绝对值，进一步合并这 4 种测试。此外，还要用一种常见的办法来处理第 2、3 两栏中出现的加法操作。综上所述，可得出下列方案：

$$x' = |x|$$
$$y' = |y|$$
$$\delta = ((x \oplus y) \overset{s}{\gg} 63) \,\&\, y'$$
$$\text{if}(x' \overset{u}{<} (y' \ll 31) + \delta)\text{则}\{\text{不会溢出}\}$$

如果在 64 位机上使用 3 条指令计算绝对值（参见 2.4 节），那么此算法只需 12 条指令，再外加一个分支语句即可。

2.14 加法、减法与乘法的特征码

许多电脑都提供了“特征码”（condition code），用以描述整数算术操作的结果。一般来说，无符号数与带符号数的加法都用一种 add 指令来做，而不论两个操作数无符号的还是带符号的（两者不能混用），特征码都能反映出操作结果的某些特性。通常可由特征码看出下列属性：

- 计算最高位后是否产生进位（如果是无符号操作，产生进位就表示溢出）。
- 如果将其视为带符号的操作，结果是否溢出。
- 若将运算结果视为一个以 2 补码表示的 32 位元带符号整数，那么忽略进位与溢出这两个因素后，它是负值、0，还是正值。

一些老式计算机还会告诉你无限精度结果（infinite precision result，也就是用 33 个位元来表示两个 32 位数的加减法操作结果）是整数、负数，还是 0。然而此特征不太容易为高级语言编译器所用，于是这种做法就不再流行了。

对于加法运算来说，这 12 种组合里只有 9 种情况可能发生。不可能出现的几种是“不进位，溢出，结果大于 0”，“不进位，溢出，结果等于 0”，“进位，溢出，结果小于 0”。这样的话，只需要 4 个位元就能表达所有特征码的值了。有两种组合非常特殊，只有特定的操作数才能引发：只有将 0 与其自身相加，才会得出“不进位，不溢出，结果等于 0”的特征码，而将负极值与其自身相加，才能出现“进位，溢出，结果等于 0”这种特征码。

对于减法操作，需要假定计算机以 $x+\bar{y}+1$ 的方式计算 $x-y$ 的值，并且进位标志的产生方式与执行加法指令相同（在这种计算方式下，“进位”在减法操作中的意思与加法刚好相反，进位标志若是 1，则表示两数之差可以容纳在 32 位元的单字中，而进位标志若是 0，反倒说明只用 1 个字组是容不下计算结果的）。于是减法操作只会出现 7 种特征码。不可能发生的特征码组合除了上面提到的 3 种外，还有两种是：“不进位，不溢出，结果为 0”，“进位，溢出，结果为 0”。

如果 CPU 的乘法器[⊖]将两个字组的乘积用双字表示，那么需要两种乘法指令：一个用于带符号数，另一个用于无符号数。（在 4 位机上执行带符号乘法，以十六进制表示，就是：$F \times F = 01$，而无符号乘法则是：$F \times F = E1$。）因为两个单字的乘积总能放在一个双字中，所以此类指令既不产生进位，也不可能溢出。

对于以单字来存放乘积（也就是取双字结果的低 32 位）的乘法指令来说，如果将操作数与运算结果视为无符号整数，且运算结果无法用一个字组容纳，那么就会产生"进位"；而若将操作数与运算结果视为带符号数，且用带符号的 2 补码无法将运算结果容纳在一个字组中时，则会发生"溢出"。于是又出现了 9 种特征码，不可能出现的那 3 种是："不进位，溢出，结果大于 0"，"不进位，溢出，结果等于 0"，"进位，不溢出，结果等于 0"。因此，将加法、减法、乘法可能出现的所有特征码一并算上，共有 10 种组合。

2.15 循环移位

下面这两个算式相当简单，让人觉得有些意外的是：它们对大于等于 0 且小于等于 32 的整数都成立，即使进行"模 32 移位"，结果也正确：

$$\text{循环左移 } n \text{ 位}: y \leftarrow (x \ll n) \mid (x \overset{u}{\gg} (32 - n))$$

$$\text{循环右移 } n \text{ 位}: y \leftarrow (x \overset{u}{\gg} n) \mid (x \ll (32 - n))$$

如果电脑支持双字长移位指令，那么可以用下述办法来实现循环移位：

```
shldi RT,RA,RB,I
shrdi RT,RA,RB,I
```

这两种指令都会把 RA 与 RB 合起来视为一个双字量，然后根据常量字段 `I` 的值对其左移或右移。（如果移位量也放在寄存器里，那么这两个指令在大部分 RISC 架构的电脑上就很难实现了，因为那样要读取 3 个寄存器。）左移操作的结果是双字节移位后的高 32 位，而右移操作的结果则是双字节移位后的低 32 位。

使用 `shldi` 指令，可以将寄存器 Rx 循环左移：

```
shldi RT,Rx,Rx,I
```

同理，循环右移也可以用 `shrdi` 指令实现。

如果只循环左移 1 位，那么可以将待循环的寄存器与其自身相加，并且算上"首尾循环进位"（end－around carry，也就是把两数求和过程中最高位所产生的进位，加到求和结果的最低位上）。大多数电脑都没有这种指令，不过在很多机子上可以用两条指令实现：（1）把寄存器的内容与其自身相加，如果产生进位，则会存放于状态寄存器中，（2）把进位加到第 1 步算出的结果上。

⊖ multiplier，一种数字电路元件，可以将两个二进制数相乘。它是由一些更基本的加法器组成的。详情参见：http：//zh.wikipedia.org/wiki/乘法器。——译者注

2.16 双字长加减法

2.13.3节介绍了一些判断无符号加减法是否溢出的式子，利用它们很容易就能在不访问CPU进位标志的前提下实现出双字长加减法。为了演示双字长加法，此处假设操作数为 $(\boldsymbol{x}_1, \boldsymbol{x}_0)$ 与 $(\boldsymbol{y}_1, \boldsymbol{y}_0)$，结果为 $(\boldsymbol{z}_1, \boldsymbol{z}_0)$。下标为1者，表示双字中权重较高的32位，为0者，表示权重较低的32位。另外还需假设寄存器的32个位元都参与运算。存放在低位字组中的量，视为无符号数。

$$
\begin{aligned}
&\boldsymbol{z}_0 \leftarrow \boldsymbol{x}_0 + \boldsymbol{y}_0 \\
&\boldsymbol{c} \leftarrow [(\boldsymbol{x}_0 \ \& \ \boldsymbol{y}_0) \mid ((\boldsymbol{x}_0 \mid \boldsymbol{y}_0) \ \& \ \neg \boldsymbol{z}_0)] \overset{u}{\gg} 31 \\
&\boldsymbol{z}_1 \leftarrow \boldsymbol{x}_1 + \boldsymbol{y}_1 + \boldsymbol{c}
\end{aligned}
$$

上述算法需要9条指令。有些电脑支持无符号数比较操作，并且能把结果以**1**或**0**的形式存放在寄存器中，比如MIPS架构CPU的“SLTU”指令[1]［MIPS］。在此类电脑上，第2行可以改写为 $\boldsymbol{c} \leftarrow (\boldsymbol{z}_0 \overset{u}{<} \boldsymbol{x}_0)$，这样只需要4个指令就够了。

双字长减法 $(\boldsymbol{x}-\boldsymbol{y})$ 也可以用类似代码实现：

$$
\begin{aligned}
&\boldsymbol{z}_0 \leftarrow \boldsymbol{x}_0 - \boldsymbol{y}_0 \\
&\boldsymbol{b} \leftarrow [(\neg \boldsymbol{x}_0 \ \& \ \boldsymbol{y}_0) \mid ((\boldsymbol{x}_0 \equiv \boldsymbol{y}_0) \ \& \ \boldsymbol{z}_0)] \overset{u}{\gg} 31 \\
&\boldsymbol{z}_1 \leftarrow \boldsymbol{x}_1 - \boldsymbol{y}_1 - \boldsymbol{b}
\end{aligned}
$$

在配有全套逻辑指令的电脑中，实现此算法共需8条指令。若CPU还支持“SLTU”指令，那么第2行可写为 $\boldsymbol{b} \leftarrow (\boldsymbol{x}_0 \overset{u}{<} \boldsymbol{y}_0)$，这样只需4条即可。

在最低有效字中，可以只用31个位元来存放数据，而让权重最高的那个位元在一般情况下置0，仅用来存放进位或借位，要是这么做的话，在大部分电脑上只用5条指令就能实现双字长加减法了。

2.17 双字长移位

设 $(\boldsymbol{x}_1, \boldsymbol{x}_0)$ 为一对尚待左移或右移的32位元字组，且 $\boldsymbol{x}_1$ 的权重较高。现将其整体视为一个64位元的量来移位，设移位结果为 $(\boldsymbol{y}_1, \boldsymbol{y}_0)$，且 $\boldsymbol{y}_1$ 的权重也较高。假定移位量 n 是取值范围在0至63之间的变量，再假定CPU的移位指令为了调整移位长度而要把它与一个量取模时，这个量必须大于等于64。意思就是说，如果对某个字组做无符号移位

[1] Set on Less Than Unsigned，该指令以无符号数的方式比较第1个源寄存器的值是否小于第2个，若是，则将目标寄存器的值设为1，否则设为0。详情参见：http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html。——译者注

时，指定的移位长度在 32 至 63 或－32 至－1 之间，那么移位结果的每一个位元值都是 0。若是带符号右移，那么移位结果中的 32 个位元，其值都等于移位前那个操作数的符号位。(本节的代码在 Intel x86 架构的电脑上无法正常运行，因为它们执行的是“模 32 移位”。)

在满足上述假设的条件后，双字左移操作可以用下面这个式子算出（需要 8 条指令）：

$$\boldsymbol{y_1 \leftarrow x_1 \ll n \mid x_0 \overset{u}{\gg} (32-n) \mid x_0 \ll (n-32)}$$

$$\boldsymbol{y_0 \leftarrow x_0 \ll n}$$

在第 1 个算式中，连接各项的操作符必须是“或”，而不能是“加”，因为只有这样做了，才能在 $n=32$ 时算出正确答案。若已知 $0 \leqslant n \leqslant 32$，那么第 1 个式子的最后一项就可以省掉，这样的话仅需 4 条指令。

同理，无符号双字右移指令的算法如下：

$$\boldsymbol{y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32-n) \mid x_1 \overset{u}{\gg} (n-32)}$$

$$\boldsymbol{y_1 \leftarrow x_1 \overset{u}{\gg} n}$$

带符号双字右移实现起来比较难，因为得把算式最后一项中那个多余的符号传播位去掉。一种较为直接的算法代码是：

$$\text{if } \boldsymbol{n < 32} \text{ then } \boldsymbol{y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32-n)}$$

$$\text{else } \boldsymbol{y_0 \leftarrow x_1 \overset{s}{\gg} (n-32)}$$

$$\boldsymbol{y_1 \leftarrow x_1 \overset{s}{\gg} n}$$

若电脑支持条件移动指令（conditional move instruction），那么很容易就能改成只需 8 条指令的无分支代码了。如果没有条件移动指令，那么可以用带符号右移 31 位的老办法构建一个掩码，然后用它把最后一项中那个多余的符号传播位屏蔽掉，这种做法共需 10 条指令。

$$\boldsymbol{y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32-n) \mid [(x_1 \overset{s}{\gg} (n-32))\ \&\ ((32-n) \overset{s}{\gg} 31)]}$$

$$\boldsymbol{y_1 \leftarrow x_1 \overset{s}{\gg} n}$$

2.18　多字节加减法与求绝对值

某些应用程序需要处理由短整数（通常是字节或半字组）构成的数组。如果每次以字组为单位运算，那么执行速度通常就能快一些。为了明确演示这个过程，我们以封装到字组中的 4 个 1 字节整数做例子。当然这种办法也很容易适配到其他封装形式上，比如封装了 1 个 12 位元整数和两个 10 位元整数的字组。在 64 位机上，这些技术的价值更大，因为能够并发执行的指令更多。

要想完成多字节加法，必须设法阻止每个字节向更高字节产生的进位。这可以用下面两个步骤完成：

1. 将每个操作数中各个字节的最高位屏蔽，然后相加（这样就不会产生跨越字节边界的进位了）。

2. 对两个操作数中每个字节的最高位做 1 位元加法，并加上由次高位带入的进位，以便修正其值。

上述第 1 步求出来的和，其最高位也就是第 2 步中说的那个由次高位带来的进位。也可以用与此相似的公式计算减法。

加法

$$s \leftarrow (x \mathbin{\&} \mathbf{0x7F7F7F7F}) + (y \mathbin{\&} \mathbf{0x7F7F7F7F})$$
$$s \leftarrow ((x \oplus y) \mathbin{\&} \mathbf{0x80808080}) \oplus s$$

减法

$$d \leftarrow (x \mid \mathbf{0x80808080}) - (y \mathbin{\&} \mathbf{0x7F7F7F7F})$$
$$d \leftarrow ((x \oplus y) \mid \mathbf{0x7F7F7F7F}) \equiv d$$

在支持全套逻辑指令的 CPU 中，如果算上加载常数 **0x7F7F 7F7F** 所需的指令，那么一共需要 8 条。（分别将式中与 **0x8080 8080** 按位与及按位或的操作替换为对 **0x7F7F 7F7F** 按位取反之后再按位与及按位或。[⊖]）

如果字组恰好分割成两个位段，那么还有一种办法实现其加减法。在此情况下，可以先做 32 位元加法，然后再把多余的进位减掉。2.13.2 节说过，可以用 $(x+y) \oplus x \oplus y$ 这个式子得出从低位带入每个位元中的进位，与之相似，还有一个适用于减法借位的式子。利用这两个式子，就可以按下述代码执行两个半字的加减法了，其结果已与 2^{16} 取模（此算法需要 7 条指令）：

加法	减法
$s \leftarrow x + y$	$d \leftarrow x - y$
$c \leftarrow (s \oplus x \oplus y) \mathbin{\&} \mathbf{0x0001\ 0000}$	$b \leftarrow (d \oplus x \oplus y) \mathbin{\&} \mathbf{0x0001\ 0000}$
$s \leftarrow s - c$	$d \leftarrow d + b$

在求多字节的绝对值时，如果某个字节表示负数（也就是其最高位是 1），那么将其按位取反再加 1 即可。下列代码会计算 x 中每个字节的绝对值，并将其放入 y 中（需 8 条指令）：

$a \leftarrow x \mathbin{\&} \mathbf{0x80808080}$	//提取每个字节的符号位
$b \leftarrow a \overset{u}{\gg} 7$	//若 x 中的相应字节为负，则 b 中的对应字节是 1
$m \leftarrow (a - b) \mid a$	//若 x 中的相应字节为负，则 m 中的对应字节是 0xFF
$y \leftarrow (x \oplus m) + b$	//若 x 中的相应字节为负，则将其按位取反再加 1

⊖ 意思是将公式中 $(x \oplus y)$ & **0x8080 8080** 替换成 $(x \oplus y)$ & ¬**0x7F7F 7F7F**，将 x | **0x8080 8080** 替换成 x | ¬**0x7F7F 7F7F**。——译者注

第 3 行也可以写成 $\boldsymbol{m} \leftarrow \boldsymbol{a}+\boldsymbol{a}-\boldsymbol{b}$。第 4 行中加 $\boldsymbol{b}$ 这个操作不会产生跨字节边界的进位，因为在 $\boldsymbol{x} \oplus \boldsymbol{m}$ 这个数中，每个字节的最高位都是 0。

2.19 doz、max、min 函数

“doz” 函数意思是“差或零”（difference or zero），其定义如下：

无符号数 带符号数

$$\mathrm{doz}(\boldsymbol{x},\boldsymbol{y})=\begin{cases}\boldsymbol{x}-\boldsymbol{y}, \boldsymbol{x} \geqslant \boldsymbol{y},\\ \boldsymbol{0}, \boldsymbol{x}<\boldsymbol{y}.\end{cases} \qquad \mathrm{dozu}(\boldsymbol{x},\boldsymbol{y})=\begin{cases}\boldsymbol{x}-\boldsymbol{y}, \boldsymbol{x} \overset{u}{\geqslant} \boldsymbol{y},\\ \boldsymbol{0}, \boldsymbol{x} \overset{u}{<} \boldsymbol{y}.\end{cases}$$

此操作也叫“一年级减法”（first grade subtraction），因为要是减去的量比被减的还多，结果就按 0 算。[㊀] 如果能以一条计算机指令来表示的话，那么最能派上用场的地方就是实现 max($\boldsymbol{x},\boldsymbol{y}$) 与 min($\boldsymbol{x},\boldsymbol{y}$) 这两个函数了（带符号版本与无符号版本均可）。大家在后面就会看到：此种情况下仅用两条指令就能实现这两个函数。很难用硬件实现 max($\boldsymbol{x}$, $\boldsymbol{y}$) 与 min($\boldsymbol{x},\boldsymbol{y}$) 函数，因为 CPU 需要一条能够绕开加法器的电路，以便把从寄存器堆（Register File）的读端口中获取到的值重新写回其输入端口。一般来说不存在这种电路。假设真有的话，那么在此区域必须排布数条接线，才能实现这种寄存器旁路。图 2.3 演示了这一情况。（求最大值或最小值的指令）用加法器（Adder）算出 $\boldsymbol{x}-\boldsymbol{y}$，然后根据减法操作的两个最高有效位（2.12.2 节中描述了判定方法）来判断 $\boldsymbol{x} \geqslant \boldsymbol{y}$ 还是 $\boldsymbol{x}<\boldsymbol{y}$。比较结果会放入一个数据选择器（MUX）[㊁] 中，然后选取器根据这一结果在 $\boldsymbol{x}$ 与 $\boldsymbol{y}$ 中选定一个值，并将其写回寄存器堆中的目标寄存器里。一般情况下并没有这种从寄存器堆里读出 $\boldsymbol{x}$、$\boldsymbol{y}$ 并实现数据选择器的电路，而且也没多大用处。实现“差或零”指令无需依靠这些电路，因为它只会把加法器的结果（或 0）直接写回寄存器堆栈，不需要数据选择器。

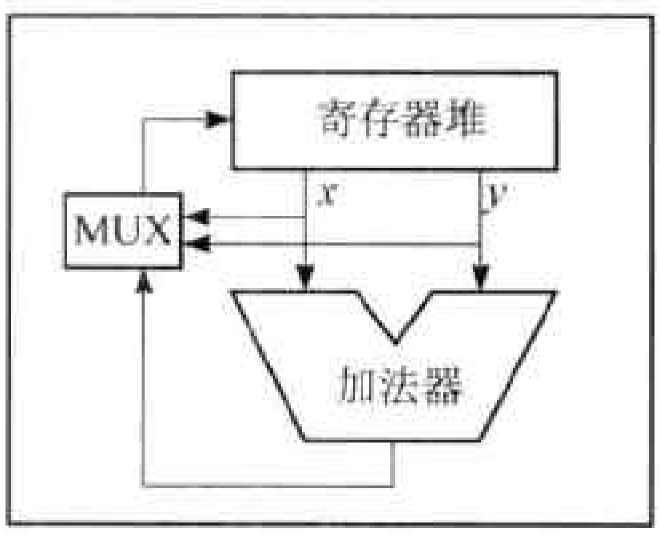


图 2.3 实现 max($\boldsymbol{x},\boldsymbol{y}$) 与 min($\boldsymbol{x},\boldsymbol{y}$) 所需的电路

用“差或零”函数实现 max($\boldsymbol{x},\boldsymbol{y}$) 与 min($\boldsymbol{x},\boldsymbol{y}$)，只需如下两条指令：

带符号数	无符号数
$\max(\boldsymbol{x},\boldsymbol{y})=\boldsymbol{y}+\mathrm{doz}(\boldsymbol{x},\boldsymbol{y})$	$\mathrm{maxu}(\boldsymbol{x},\boldsymbol{y})=\boldsymbol{y}+\mathrm{dozu}(\boldsymbol{x},\boldsymbol{y})$
$\min(\boldsymbol{x},\boldsymbol{y})=\boldsymbol{x}-\mathrm{doz}(\boldsymbol{x},\boldsymbol{y})$	$\mathrm{minu}(\boldsymbol{x},\boldsymbol{y})=\boldsymbol{x}-\mathrm{dozu}(\boldsymbol{x},\boldsymbol{y})$

㊀ 数学家把这种运算叫做 monus，用符号 $\dot{-}$ 来表示。此操作也称“正值差”（positive difference）或“饱和减法”（saturated subtraction）。

㊁ multiplexor，是一种可以从多个输入信号中选择一个信号输出的器件。详情参见：http://zh.wikipedia.org/wiki/数据选择器。——译者注

在带符号数的情况下，如果 $x-y$ 溢出了，那么“差或零”指令的结果就是负数。这里的溢出可以忽略，因为把该值与 y 相加或者从 x 里减去，都会导致再次溢出，于是两次溢出之后的结果反而是正确的。其中的原因是：如果 $\mathrm{doz}(x,y)$ 是负的，那么只需把它视为无符号数，即可正确解读出两者的差了。

要是想在没有“差或零”指令的电脑上以无分支的方式快速实现 $\mathrm{doz}(x,y)$、$\max(x,y)$等函数的话，可以参考下面几段。我们将会分别演示如何在具有条件移动指令、比较谓词，以及能够高效访问进位标志的 CPU 上实现这些函数，并且还会告诉大家怎样在不具备上述功能的电脑上实现它。

若是电脑支持条件移动指令，那么用 3 条指令就能实现 $\mathrm{doz}(x,y)$ 函数。以破坏原始参数的方式[⊖]实现 $\max(x,y)$ 与 $\min(x,y)$ 函数，需要两条指令。举例来说，在具备完整 RISC 指令集的计算机上可用如下指令（其中的 r0 表示值恒为 0 的寄存器）实现 $z\leftarrow\mathrm{doz}(x,y)$：

```
sub     z,x,y       将 z 设为 x - y
cmplt   t,x,y       如果 x<y，就将 t 设为 1，否则设为 0
movne   z,t,r0      如果 x<y，就将 z 设为 0
```

在具备完整 RISC 指令集的电脑中，$x\leftarrow\max(x,y)$ 可以用下列指令实现：

```
cmplt   t,x,y       如果 x<y，就将 t 设为 1，否则设为 0
movne   x,t,y       如果 x<y，就将 y 赋给 x
```

只需改换上述各指令的比较条件，就可以实现出 min 函数以及这几个函数的无符号版本。

通过比较谓词来实现这些函数，需要 4 至 5 条指令（如果比较谓词用 -1 表示“真”，则需要的指令数就是 3 至 4 条）：

$$
\begin{aligned}
\mathrm{doz}(x,y) &= (x-y)\ \&\ -(x\geqslant y)\\
\max(x,y) &= y+\mathrm{doz}(x,y)\\
&= ((x\oplus y)\ \&\ -(x\geqslant y))\oplus y\\
\min(x,y) &= x-\mathrm{doz}(x,y)\\
&= ((x\oplus y)\ \&\ -(x\leqslant y))\oplus y
\end{aligned}
$$

在某些电脑上，可以依靠借位标志来计算这些函数的无符号版本。设 $\mathrm{carry}(x-y)$ 为加法器在计算 $x+\bar{y}+1$ 之后产生并存放于寄存器中的进位。于是，当且仅当 $x\geqslant y$ 时，$\mathrm{carry}(x-y)$ 才可能等于 1。由此可得：

$$
\begin{aligned}
\mathrm{dozu}(x,y) &= ((x-y)\ \&\ \neg(\mathrm{carry}(x-y)-1))\\
\mathrm{maxu}(x,y) &= x-((x-y)\ \&\ (\mathrm{carry}(x-y)-1))\\
\mathrm{minu}(x,y) &= y+((x-y)\ \&\ (\mathrm{carry}(x-y)-1))
\end{aligned}
$$

有些机子具备一种能够产生进位或借位的减法，同时还提供另外一种把上述指令产

⊖ destructive operation，破坏性操作，此类操作会修改其一个或多个参数的值。

生的进位或借位当做输入值的减法。在具备这两种指令的大部分电脑中，只要计算完 $\boldsymbol{x}-\boldsymbol{y}$ 之后再加一条指令，即可求出 carry$(\boldsymbol{x}-\boldsymbol{y})-1$ 的值。举例来说，在 Intel x86 平台上，minu$(\boldsymbol{x},\boldsymbol{y})$ 可以用如下 4 条指令计算：

```
sub eax, ecx    ; 该指令的输入值 x 与 y 分别存放于 eax 及 ecx 寄存器中
sbb edx, edx    ; 如果 x> = y，那么 edx = 0，否则为 - 1
and eax, edx    ; 如果 x> = y，那么结果为 0，否则为 x - y
add eax, ecx    ; 将上条指令的结果加 y。若 x> = y，那么结果为 y，否则为 x
```

依此方式，这 3 个函数都可以用 4 条指令实现出来（要是电脑中有“按位取反后按位与”这个指令，那么 dozu$(\boldsymbol{x},\boldsymbol{y})$ 只需 3 条）。

有一种办法几乎适用于任何 RISC 架构的 CPU，那就是在上述各种表达式中挑选一种使用比较谓词的算法，然后用 2.12 节所讲的任意一种办法替换掉比较谓词。例如：

$$\boldsymbol{d} \leftarrow \boldsymbol{x}-\boldsymbol{y}$$

$$\mathrm{doz}(\boldsymbol{x},\boldsymbol{y})=\boldsymbol{d}\ \&\left[(\boldsymbol{d} \equiv ((\boldsymbol{x} \oplus \boldsymbol{y}) \& (\boldsymbol{d} \oplus \boldsymbol{x}))) \overset{s}{\gg} 31\right]$$

$$\mathrm{dozu}(\boldsymbol{x},\boldsymbol{y})=\boldsymbol{d}\ \& \neg\left[((\neg \boldsymbol{x}\ \& \boldsymbol{y}) \mid ((\boldsymbol{x} \equiv \boldsymbol{y}) \& \boldsymbol{d})) \overset{s}{\gg} 31\right]$$

根据电脑指令集的丰富程度，上述算法需要 7 至 10 条指令。若要实现 max 或 min 函数，则所需指令数还要再加 1。

若已知 $-2^{31} \leqslant x-y \leqslant 2^{31}-1$（此处为普通算术，而非计算机算术），那么只用 4 条无分支的基本 RISC 指令就可以实现这些函数了。只要 x 与 y 的差满足这个关系式，不论是带符号整数还是无符号整数，都可以用相同的代码实现。对于带符号整数来说，下列公式成立的充分条件（sufficient condition）是 $-2^{30} \leqslant x$，$y \leqslant 2^{30}-1$，而对于无符号数，则是 $0 \leqslant x$，$y \leqslant 2^{31}-1$。

$$\mathrm{doz}(\boldsymbol{x},\boldsymbol{y})=\mathrm{dozu}(\boldsymbol{x},\boldsymbol{y})=(\boldsymbol{x}-\boldsymbol{y}) \& \neg((\boldsymbol{x}-\boldsymbol{y}) \overset{s}{\gg} 31)$$

$$\max(\boldsymbol{x},\boldsymbol{y})=\mathrm{maxu}(\boldsymbol{x},\boldsymbol{y})=\boldsymbol{x}-((\boldsymbol{x}-\boldsymbol{y}) \& ((\boldsymbol{x}-\boldsymbol{y}) \overset{s}{\gg} 31))$$

$$\min(\boldsymbol{x},\boldsymbol{y})=\mathrm{minu}(\boldsymbol{x},\boldsymbol{y})=\boldsymbol{y}+((\boldsymbol{x}-\boldsymbol{y}) \& ((\boldsymbol{x}-\boldsymbol{y}) \overset{s}{\gg} 31))$$

下面列举一些“差或零”指令的用途。在这些情况下，doz$(\boldsymbol{x},\boldsymbol{y})$ 的结果必须视为无符号整数。

1. 直接实现 Fortran 语言的 IDIM 函数。

2. 计算两数之差的绝对值 [Knu7]：

$$\begin{aligned} |\boldsymbol{x}-\boldsymbol{y}| &= \mathrm{doz}(\boldsymbol{x},\boldsymbol{y})+\mathrm{doz}(\boldsymbol{y},\boldsymbol{x}) && \text{适用于无符号数} \\ &= \mathrm{dozu}(\boldsymbol{x},\boldsymbol{y})+\mathrm{dozu}(\boldsymbol{y},\boldsymbol{x}) && \text{适用于带符号数} \end{aligned}$$

推论：$|\boldsymbol{x}|=\mathrm{doz}(\boldsymbol{x},\boldsymbol{0})+\mathrm{doz}(\boldsymbol{0},\boldsymbol{x})$（2.4 节列出了另外一些用 3 条指令来求绝对值的方法）。

3. 若两个无符号整数 $\boldsymbol{x}$ 与 $\boldsymbol{y}$ 之和超过了 32 位元所能表达的最大正整数（也就是 $2^{32}-1$），那么将求和结果强行下调为（clamp）该值 [Knu7]：

$$\neg \mathrm{dozu}(\neg \boldsymbol{x}, \boldsymbol{y})$$

4. 实现某些比较谓词（各需4条指令）：

$$\boldsymbol{x} > \boldsymbol{y} = (\mathrm{doz}(\boldsymbol{x}, \boldsymbol{y}) \mid -\mathrm{doz}(\boldsymbol{x}, \boldsymbol{y})) \overset{u}{\gg} 31$$

$$\boldsymbol{x} \overset{u}{>} \boldsymbol{y} = (\mathrm{dozu}(\boldsymbol{x}, \boldsymbol{y}) \mid -\mathrm{dozu}(\boldsymbol{x}, \boldsymbol{y})) \overset{u}{\gg} 31$$

5. 计算由加法 $\boldsymbol{x}+\boldsymbol{y}$ 产生的进位（需5条指令）：

$$\mathrm{carry}(\boldsymbol{x}+\boldsymbol{y}) = \boldsymbol{x} \overset{u}{>} \neg \boldsymbol{y} = (\mathrm{dozu}(\boldsymbol{x}, \neg \boldsymbol{y}) \mid -\mathrm{dozu}(\boldsymbol{x}, \neg \boldsymbol{y})) \overset{u}{\gg} 31$$

如果将表达式 $\mathrm{doz}(\boldsymbol{x}, -\boldsymbol{y})$ 的值视为无符号整数，那么在大多数情况下它就能代表 $\boldsymbol{x}+\boldsymbol{y}$ 的和，只是低于0的结果会被强行上调为0。然而若 $\boldsymbol{y}$ 是负极值 $\boldsymbol{x}$ 不是负极值，则此方法无效。

IBM RS/6000电脑及其前身801[⊖]具有带符号的“差或零”指令，而Knuth的MMIX计算机[⊜][Knu7]支持无符号版本的指令（包含一些派生指令，它们能够并发操作字组中某些部分）。于是这就产生一个问题：如何用无符号版本的指令来实现带符号的指令，反之亦然。可以用下列公式来转换这两个指令（其中的加减法只是为了反转符号位而已）：

$$\mathrm{doz}(\boldsymbol{x}, \boldsymbol{y}) = \mathrm{dozu}(\boldsymbol{x}+2^{31}, \boldsymbol{y}+2^{31})$$

$$\mathrm{dozu}(\boldsymbol{x}, \boldsymbol{y}) = \mathrm{doz}(\boldsymbol{x}-2^{31}, \boldsymbol{y}-2^{31})$$

还有两个有用的等式：

$$\mathrm{doz}(\neg \boldsymbol{x}, \neg \boldsymbol{y}) = \mathrm{doz}(\boldsymbol{y}, \boldsymbol{x})$$

$$\mathrm{dozu}(\neg \boldsymbol{x}, \neg \boldsymbol{y}) = \mathrm{dozu}(\boldsymbol{y}, \boldsymbol{x})$$

如果 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 里面有一个是负极值而另一个不是，那么关系式 $\mathrm{doz}(\neg \boldsymbol{x}, \neg \boldsymbol{y}) = \mathrm{doz}(\boldsymbol{y}, \boldsymbol{x})$ 就不成立了。

2.20 互换寄存器中的值

有一个流传相当久的技巧，可以在不使用第3个寄存器的前提下交换两个寄存器的内容[IBM]：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus \boldsymbol{y}$$

$$\boldsymbol{y} \leftarrow \boldsymbol{y} \oplus \boldsymbol{x}$$

$$\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus \boldsymbol{y}$$

这个技巧很适用于那种指令里只能使用两个地址的计算机（two-address machine）。也可以用≡逻辑操作（先按位异或再按位取反）来代替⊕操作，而且还能用多种形式的

⊖ 是一项IBM于1974年启动的研究计划，其间产生了RISC指令集的概念。详情参见：http://zh.wikipedia.org/wiki/IBM_POWER。——译者注

⊜ MMIX是高德纳教授为了系统地讲解其算法理论而构想的一台计算机，采用64位RISC架构，便于描述实现算法所用的汇编语言指令。详情参见：http://en.wikipedia.org/wiki/MMIX。——译者注

加减法算式来改写：

$$
\begin{array}{lll}
x \leftarrow x+y & x \leftarrow x-y & x \leftarrow y-x \\
y \leftarrow x-y & y \leftarrow y+x & y \leftarrow y-x \\
x \leftarrow x-y & x \leftarrow y-x & x \leftarrow x+y
\end{array}
$$

不巧的是，上面这些通过加减法来交换寄存器的算法中，都包含一条不适用于二地址计算机的指令[㊀]，除非这些电脑上有“逆向减法”[㊁]。

这项小技巧在编写使用双缓冲功能的应用程序时很有用，这种程序要交换两个指针。我们把交换算法的第一条指令提出来，放在执行交换操作的那个循环前面（这样做要多用一个寄存器，没办法像原算法那样只用两个寄存器）：

循环外：$t \leftarrow x \oplus y$

循环内：$x \leftarrow x \oplus t$

$y \leftarrow y \oplus t$

2.20.1　交换寄存器中相应的位段

这里研究的问题是根据掩码 m 来交换两个寄存器 x、y 中的内容：如果掩码中的某位 $m_i=1$，那就交换 x 与 y 对应位元上的值，如果 $m_i=0$，那么这两个位元就保持不变。这里强调“相应”位段的意思是，在不移位的前提下完成操作。m 中值为 1 的位元不需要连续出现。一种较为直接的算法如下：

$$
\begin{aligned}
&x' \leftarrow (x \,\&\bar{m}) \mid (y \,\&m) \\
&y \leftarrow (y \,\&\bar{m}) \mid (x \,\&m) \\
&x \leftarrow x'
\end{aligned}
$$

如果用“临时寄存器”来保存 4 个按位与表达式的运算结果，并假设加载 m 或 $\bar{m}$ 都用一条指令，计算机执行“按位取反后按位与”操作也只需一条指令，那么整个算法用 7 条指令即可实现。要是电脑能够并行计算 4 个按位与表达式，那么此算法仅需 3 个周期就能执行完。

下表中栏（a）所列的方法可能更好些（仅需 5 条指令，不过在具备无限指令级并行能力的计算机上要花 4 个周期执行），它是根据前面那个交换寄存器内容所用的“三次求异或”代码推演而来。

(a)	(b)	(c)
$x \leftarrow x \oplus y$	$x \leftarrow x \equiv y$	$t \leftarrow (x \oplus y) \,\& m$
$y \leftarrow y \oplus (x \,\&m)$	$y \leftarrow y \equiv (x \mid \bar{m})$	$x \leftarrow x \oplus t$
$x \leftarrow x \oplus y$	$x \leftarrow x \equiv y$	$y \leftarrow y \oplus t$

㊀ 指目标寄存器与被加数或被减数不符的那三条：$y \leftarrow x-y$，$x \leftarrow y-x$，$x \leftarrow x+y$。——译者注

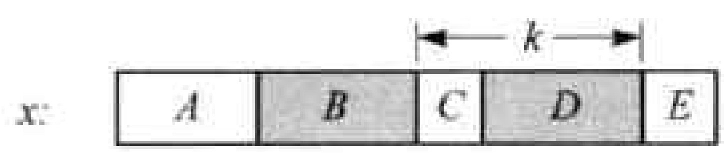
㊁ 一般减法指令中，先出现的寄存器表示被减数，后出现的表示减数，而“逆向减法”（reverse subtract）与之相反。——译者注

（b）栏所列算法与（a）栏一样，都可以交换寄存器的内容，然而如果 $\boldsymbol{m}$ 的值无法容纳于指令码的常量字段（immediate field）中，但 $\bar{\boldsymbol{m}}$ 却可以，并且计算机还支持按位等值指令的话，那么这种算法就很有用了。

（c）栏中又列出了一种算法［GLS1］，也需要5条指令（还得假设计算机只用1条指令就能把 $\boldsymbol{m}$ 载入寄存器），然而在一台指令级并行能力足够好的计算机上，仅用3个周期即可执行完。

2.20.2 交换同一寄存器内的两个位段

假设我们想在不改变寄存器中其他位元值的前提下，交换 $\boldsymbol{x}$ 中两个长度相同的位段。也就是说，在下图所描述的计算机字组中，要在不改变 A、C、E 的前提下，交换 B 与 D。两个待交换位段之间的距离是 k。



较为直接的实现方式是将 D 与 B 分别移动到新的位置，然后再用按位与及按位或操作将字组中的各个部件拼合连接起来。其算法如下：

$$\boldsymbol{t}_1 = (\boldsymbol{x} \,\&\, \boldsymbol{m}) \ll k$$
$$\boldsymbol{t}_2 = (\boldsymbol{x} \overset{u}{\gg} k) \,\&\, \boldsymbol{m}$$
$$\boldsymbol{x}' = (\boldsymbol{x} \,\&\, \boldsymbol{m}') \mid \boldsymbol{t}_1 \mid \boldsymbol{t}_2$$

算式中的 $\boldsymbol{m}$ 是个掩码，如果位段 D 的某个位元是1，则 $\boldsymbol{m}$ 的对应位元也是1（其余位元都是0），$\boldsymbol{m}'$ 也是个掩码，其中值为1的位元，对应位段 A、C、D 中的相应位置上也为1的那些位元。在指令级并行能力不受限的计算机上，如果算上生成两个掩码所用的4条指令，那么实现这些式子共需11条指令，执行起来要花6个周期。

还有一种办法［GLS1］也基于上文所说的那些限定条件，然而只需8条指令即可实现，而且执行时间也只有5个周期。它与2.20.1节（c）栏中交换两个寄存器对应位段所用的代码类似。这里的 $\boldsymbol{m}$ 也是个掩码，用于析出位段 D 中值为1的位元。

$$\boldsymbol{t}_1 = [\boldsymbol{x} \oplus (\boldsymbol{x} \overset{u}{\gg} k)] \,\&\, \boldsymbol{m}$$
$$\boldsymbol{t}_2 = \boldsymbol{t}_1 \ll k$$
$$\boldsymbol{x}' = \boldsymbol{x} \oplus \boldsymbol{t}_1 \oplus \boldsymbol{t}_2$$

这个算法的思路是把 $B \oplus D$ 的值放在位段 D 所处的位置上（其余位置填0），并用 $\boldsymbol{t}_1$ 表示它，再把同一个值又放在位段 B 所处的位置上，并用 $\boldsymbol{t}_2$ 表示它。此代码与早前那段代码一样，只有当 B 与 D 是两个“相互分离的位段”（split fields）时才能算出正确结果，换句话说，掩码 $\boldsymbol{m}$ 中不能只出现一串值为1的位元。[⊖]

⊖ 意思就是，必须出现两串相互分离且值为1的位元。也可以理解成位段 B 与 D 的间距 k 必须大于0。——译者注

2.20.3 有条件的交换

在前两小节中，若 $\boldsymbol{m}$ 为 0，则那些基于按位异或的算法就退化成了没有任何效果的空操作（no-operation）。因此，如果我们在某条件 c 为**真**时，将掩码 $\boldsymbol{m}$ 中相应的位元设置成 1，而在条件 c 为**假**时，把 $\boldsymbol{m}$ 置为 0，那么就可以有条件地互换两个寄存器、两个寄存器中对应的位段，以及同一寄存器内两个位段的内容了。假如 $\boldsymbol{m}$ 的值能够用无分支代码来设定，那么整个条件式交换算法也就是无分支的。

2.21 在两个或两个以上的值之间切换

假设变量 x 的取值只可能是 a 或 b，而你需要将 x 设为与当前值不同的另外一个值，并且想让代码逻辑不受制于 a 和 b 的具体值。举例来说，在某个编译器中，x 可能表示一个操作码，它要么是 branch true，要么是 branch false[⊖]，而不管 x 的当前值是哪一个，我们都想把它的值设置成另外一个。这里说的 branch true 与 branch false 操作码是随意举的例子，它们有可能是在 C 语言头文件中用 `#define` 或 `enum` 声明的。

在两个值之间切换，一种较为直接的代码是：

```
if (x == a) x = b;
else x = a;
```

或是像 C 程序中常见的那样：

```
x = x == a ? b : a;
```

下面两种做法远比上面两种好（至少可以说效率比较高一些）：

$$\boldsymbol{x} \leftarrow \boldsymbol{a} + \boldsymbol{b} - \boldsymbol{x}$$
$$\boldsymbol{x} \leftarrow \boldsymbol{a} \oplus \boldsymbol{b} \oplus \boldsymbol{x}$$

若 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 为常数，则上述算法只需 1 至 2 条基本 RISC 指令。当然了，此处无需理会 $\boldsymbol{a}+\boldsymbol{b}$ 可能产生的溢出。

于是这就冒出来一个问题：有没有一种能在 3 个或 3 个以上的值之间轮替的高效算法？换句话说，给定 3 个互不相同的常数 a、b、c，能不能找到一个满足下列条件且易于求值的函数 f：

$$f(a)=b$$
$$f(b)=c$$
$$f(c)=a$$

说来有趣，其实这儿就有个现成的多项式能表达此函数。对于三个常数的情况，这个式子就是：

⊖ 这两条指令分别表示在判断条件成立时转入分支，以及在判断条件不成立时转入分支。——译者注

$$f(x)=\frac{(x-a)(x-b)}{(c-a)(c-b)}a+\frac{(x-b)(x-c)}{(a-b)(a-c)}b+\frac{(x-c)(x-a)}{(b-c)(b-a)}c \tag{5}$$

（构建这个函数时所用的思路是：若 $x=a$，则第一项与最后一项消失，中间一项则化简为 b，依此类推。）求该式的值需要14次算术操作，而且如果 a、b、c 为任意值，那么中间运算的结果已经无法容纳于32位宽的字组中了。不过没关系，它只不过是个二次函数，所以可先将其展开为普通的二项式，然后再利用霍纳法则[⊖]来求值，这样只要5次算术操作就够了（其中4个操作用于计算整系数的二次项，再用1条计算最后的除法）。于是可将等式（5）改写成：

$$\begin{aligned}f(x)=\frac{1}{(a-b)(a-c)(b-c)}\{&[(a-b)a+(b-c)b+(c-a)c]x^2\\&+[(a-b)b^2+(b-c)c^2+(c-a)a^2]x\\&+[(a-b)a^2b+(b-c)b^2c+(c-a)ac^2]\}\end{aligned}$$

这种写法既无趣，又无用。

另外一种方法与等式（5）类似，它的3项中也只有一项能保留下来：

$$f(\boldsymbol{x})=((-(\boldsymbol{x}=\boldsymbol{c}))\&\boldsymbol{a})+((-(\boldsymbol{x}=\boldsymbol{a}))\&\boldsymbol{b})+((-(\boldsymbol{x}=\boldsymbol{b}))\&\boldsymbol{c})$$

在具备相等谓词（equal predicate）的计算机上，不算加载常数所费的指令，实现此算法共需11条。由于最后一步的两个加法会把两个零值与一个非零值相加，所以加号可以用按位或、按位异或操作代替。

如果能预先求出 $\boldsymbol{a}-\boldsymbol{c}$ 与 $\boldsymbol{b}-\boldsymbol{c}$ 的值，那么就可以将公式简化为下述算法［GLS1］：

$$f(\boldsymbol{x})=((-(\boldsymbol{x}=\boldsymbol{c}))\&(\boldsymbol{a}-\boldsymbol{c}))+((-(\boldsymbol{x}=\boldsymbol{a}))\&(\boldsymbol{b}-\boldsymbol{c}))+\boldsymbol{c}\text{，或}$$

$$f(\boldsymbol{x})=((-(\boldsymbol{x}=\boldsymbol{c}))\&(\boldsymbol{a}\oplus\boldsymbol{c}))\oplus((-(\boldsymbol{x}=\boldsymbol{a})\&(\boldsymbol{b}\oplus\boldsymbol{c}))\oplus\boldsymbol{c}$$

上述两种方法各需8条指令。然而在大部分电脑中，它们都不如下面这种直接以C语言代码写出来的算法，因为在a、b、c的值比较小的情况下，后者只需4至6条指令。

```
if (x == a) x = b;
else if (x == b) x = c;
else x = a;
```

要是还想深究这个问题的话，这里有一种巧妙的无分支算法［GLS1］，能够在没有比较谓词指令的电脑中实现3值轮替。在大部分计算机上，该算法需要8条指令。

由于 $\boldsymbol{a}$、$\boldsymbol{b}$、$\boldsymbol{c}$ 互不相同，那么会有两个位置 n_1 与 n_2，$\boldsymbol{a}$、$\boldsymbol{b}$、$\boldsymbol{c}$ 三者在这两个位置上的位元值不一致，而且，在 n_1 位置上，其位元值与其他两数相异的数，与在 n_2 位置上，其位元值与其他两数相异的数，肯定不是同一个。下图以二进制形式演示了三数分别为21、31与20的情况：

⊖ 霍纳法则的主旨就是提取 x，比如，对一个四次多项式 $ax^4+bx^3+cx^2+dx+e$ 应用霍纳法则，结果就是 $x(x(x(ax+b)+c)+d)+e$。这样的话，一个 n 次多项式，就可以用 n 次乘法与 n 次加法算出来了。于是，那些支持乘积累加指令的计算机，实现该算法就非常方便了。

1	0	1	0	1	$\boldsymbol{c}$
1	1	1	1	1	$\boldsymbol{a}$
1	0	1	0	0	$\boldsymbol{b}$
	n_1			n_2	

为了使推导过程更通用，我们按照上图所示重排 $\boldsymbol{a}$、$\boldsymbol{b}$、$\boldsymbol{c}$ 三者的顺序，让 n_1 位置上的“特殊数字”是 $\boldsymbol{a}$，而 n_2 位置上的特殊数字是 $\boldsymbol{b}$。于是，在三数轮替过程中，n_1 位置上可能出现的位元排列形式只有两种，$(\boldsymbol{a}_{n1}, \boldsymbol{b}_{n1}, \boldsymbol{c}_{n1}) = (0, 1, 1)$ 或 $(1, 0, 0)$。同理，n_2 位置上可能出现的位元排列形式也只有两种，$(\boldsymbol{a}_{n2}, \boldsymbol{b}_{n2}, \boldsymbol{c}_{n2}) = (0, 1, 0)$ 或 $(1, 0, 1)$。于是共有 4 种情况，适用于每种情况的公式如下：

情况 1：$(\boldsymbol{a}_{n1},\boldsymbol{b}_{n1},\boldsymbol{c}_{n1})=(0,1,1),(\boldsymbol{a}_{n2},\boldsymbol{b}_{n2},\boldsymbol{c}_{n2})=(0,1,0)$：

$$f(\boldsymbol{x})=\boldsymbol{x}_{n1}*(\boldsymbol{a}-\boldsymbol{b})+\boldsymbol{x}_{n2}*(\boldsymbol{c}-\boldsymbol{a})+\boldsymbol{b}$$

情况 2：$(\boldsymbol{a}_{n1},\boldsymbol{b}_{n1},\boldsymbol{c}_{n1})=(0,1,1),(\boldsymbol{a}_{n2},\boldsymbol{b}_{n2},\boldsymbol{c}_{n2})=(1,0,1)$：

$$f(\boldsymbol{x})=\boldsymbol{x}_{n1}*(\boldsymbol{a}-\boldsymbol{b})+\boldsymbol{x}_{n2}*(\boldsymbol{a}-\boldsymbol{c})+(\boldsymbol{b}+\boldsymbol{c}-\boldsymbol{a})$$

情况 3：$(\boldsymbol{a}_{n1},\boldsymbol{b}_{n1},\boldsymbol{c}_{n1})=(1,0,0),(\boldsymbol{a}_{n2},\boldsymbol{b}_{n2},\boldsymbol{c}_{n2})=(0,1,0)$：

$$f(\boldsymbol{x})=\boldsymbol{x}_{n1}*(\boldsymbol{b}-\boldsymbol{a})+\boldsymbol{x}_{n2}*(\boldsymbol{c}-\boldsymbol{a})+\boldsymbol{a}$$

情况 4：$(\boldsymbol{a}_{n1},\boldsymbol{b}_{n1},\boldsymbol{c}_{n1})=(1,0,0),(\boldsymbol{a}_{n2},\boldsymbol{b}_{n2},\boldsymbol{c}_{n2})=(1,0,1)$：

$$f(\boldsymbol{x})=\boldsymbol{x}_{n1}*(\boldsymbol{b}-\boldsymbol{a})+\boldsymbol{x}_{n2}*(\boldsymbol{a}-\boldsymbol{c})+\boldsymbol{c}$$

在上述公式中，每个乘法操作的左操作数都只有一个二进制位，这种与 0 或 1 相乘的操作，可以替换为同所有位元均是 0 或均是 1 的数求按位与。这样的话，上述第一种情况下的公式就可改写为：

$$f(\boldsymbol{x})=(((\boldsymbol{x}\ll(31-n_1))\overset{s}{\gg}31)\,\&\,(\boldsymbol{a}-\boldsymbol{b}))+(((\boldsymbol{x}\ll(31-n_2))\overset{s}{\gg}31)\,\&\,(\boldsymbol{c}-\boldsymbol{a}))+\boldsymbol{b}$$

由于除了变量 $\boldsymbol{x}$ 之外的值均为常数，所以这个式子只需 8 个基本 RISC 指令就能实现。这里的加减法操作也可以替换成按位异或。

此算法的思路可推广到切换 4 个或更多个常数值的操作上，其要旨在于：找到能够区分各常数的一系列位置 n_1，n_2，…，使这些数在这一系列位置上的位元组合各不相同㊀。对于 4 常数轮替的情况，找到 3 个这样的位置就够了。然后（以 4 个常数切换为例），求出下列方程中 s、t、u、v 的值（也就是求下面这个一元四次方程在系数 x_{n_i} 为 0 或 1 时的解，使得 $f(x)$ 分别等于 a、b、c、d）：

$$f(x) = x_{n_1}s + x_{n_2}t + x_{n_3}u + v$$

㊀ 例如，若 4 个无符号数分别为 0（二进制 0000），1（二进制 0001），4（二进制 0100），9（二进制 1001），则需要选择 $n_1=3$，$n_2=2$，$n_3=0$ 这三个位置，因为只有这样，才能以 4 个数字在这 3 个位置上的位元组合（分别是 000，001，010，101）把它们彻底区分开。如果只选取 n_1、n_2 两个位置，就没办法把这 4 个数区隔开，因为 0（二进制 0000）和 1（二进制 0001）在这两个位置上的位元组合都是“00”。——译者注

假如只选取两个位置就能把这 4 个常数区隔开[1]，那么待求解的则是下面这个式子：

$$f(x) = x_{n_1}s + x_{n_2}t + x_{n_1}x_{n_2}u + v$$

2.22 布尔函数分解公式

本节我们来看看：想实现一个自变量个数为 3、4 或 5 的布尔函数（Boolean function），最少需要几个二元布尔操作（Boolean operation）或布尔指令才行。此处的“布尔函数”是指那种自变量为布尔值且函数值亦为布尔值的函数。

在本书所列的布尔代数算式中，“+”表示逻辑或，两数相邻表示求逻辑与，$\oplus$表示逻辑异或，上划线或¬前缀代表逻辑非。这些操作符都可以用“单比特操作数”（single-bit operand）或计算机字组的按位操作表示出来。分解布尔函数主要依靠下述定理：

定理 若 $f(x,y,z)$ 是有三个自变量的布尔函数，则其可分解为 $g(x,y) \oplus zh(x,y)$ 的形式。其中 g 与 h 皆为带两个自变量的布尔函数。[2]

证明 [Ditlow]：将 $f(x,y,z)$ 展开为两个小项（minterm）之和，并把 $\bar{z}$ 与 z 分别提取至每项前面，由此可得：

$$f(x,y,z) = \bar{z}f_0(x,y) + zf_1(x,y)$$

由于“+”的两个操作数不可能同时为 1，所以可用逻辑异或操作代替逻辑或，于是：

$$\begin{aligned} f(x,y,z) &= \bar{z}f_0(x,y) \oplus zf_1(x,y) \\ &= (1 \oplus z)f_0(x,y) \oplus zf_1(x,y) \\ &= f_0(x,y) \oplus zf_0(x,y) \oplus zf_1(x,y) \\ &= f_0(x,y) \oplus z(f_0(x,y) \oplus f_1(x,y)) \end{aligned}$$

上述推导过程中两次用到恒等式 $(a \oplus b)c = ac \oplus bc$。 ■

在上式中，$f_0(x,y)$ 就是定理中要求的 $g(x,y)$，而 $f_0(x,y) \oplus f_1(x,y)$ 也就是定理中说的那个 $h(x,y)$。顺便说一句，将 $z=0$ 代入 $f(x,y,z)$，即可求出函数 $f_0(x,y)$；而将 $z=1$ 代入 $f(x,y,z)$，即可求出函数 $f_1(x,y)$。

推论 若 16 种双变量布尔函数都能用计算机指令集中的一条指令来实现，则任意三变量布尔函数都可用 4 条（或更少的）指令实现。

在这 4 条指令中，一条用于计算 $g(x,y)$，另一条用来计算 $h(x,y)$，然后再用逻辑

---

[1] 比如 4 个无符号数 0（二进制 000）、5（二进制 101）、2（二进制 010）、7（二进制 111），只需选取 $n_1=1$，$n_2=0$ 即可用 4 个数在这两个位置上的位元组合（分别是 00、01、10、11）将其分隔开。——译者注

[2] 逻辑电路设计者把这叫做 Reed-Muller Decomposition，亦称正 Davio 分解（Positive Davio Decomposition）。根据高德纳先生在 [Knu4，7.1.1] 中的说法，此式因 I. I. Zhegalkin 而出名 [Matematicheskii Sbornik 35 (1928)，311-369]，有时也称为“俄式分解法”（Russian Decomposition）。

与、逻辑异或指令将其组合起来即可。

我们用下面这个布尔函数来演示一下。在 x、y、z 三值中，如果有两个值都是 1，而另外一个不是，那么函数值就是 1，否则函数值为 0：

$$f(x,y,z) = xy\bar{z} + x\bar{y}z + \bar{x}yz$$

在继续往下看之前，有兴趣的读者可以试试在不依赖前述定理的情况下，自己用 4 条指令把这个函数实现出来。

从定理的证明过程中可知：

$$\begin{aligned} f(x,y,z) &= f_0(x,y) \oplus z(f_0(x,y) \oplus f_1(x,y)) \\ &= xy \oplus z(xy \oplus (x\bar{y} + \bar{x}y)) \\ &= xy \oplus z(x + y) \end{aligned}$$

于是，只用 4 条指令即可实现该函数。

显然，定理也可以推广到自变量个数为 4 或更多的函数上。也就是说，任意布尔函数 $f(x_1,x_2,\cdots,x_n)$ 都可以分解为 $g(x_1,x_2,\cdots,x_{n-1}) \oplus x_n h(x_1,x_2,\cdots,x_{n-1})$。因此，四变量布尔函数可按如下步骤分解：

$$f(w,x,y,z)=g(w,x,y) \oplus zh(w,x,y),\text{其中}$$
$$g(w,x,y)=g_1(w,x) \oplus yh_1(w,x),\text{而}$$
$$h(w,x,y)=g_2(w,x) \oplus yh_2(w,x)。$$

由此可以看出，若 16 种双变量布尔函数都可以用 1 条指令实现，那么任意四变量的布尔函数都可用 10 条指令实现，同理，任意五变量的布尔函数都可用 22 条指令实现[⊖]。

然而实际上用不了这么多条指令。对于自变量个数为 4 或大于 4 的布尔函数来说，也许不能像上面这样用定理找到简单的分解式，不过已经有人用计算机彻底研究了这个问题。研究表明：任意四变量布尔函数都只需 7 条二元布尔指令实现。而任意五变量的布尔函数只需 12 条指令即可实现 [Knu4，7.1.2]。

在 2^{2^5} 个（也就是 4 294 967 296 个）五变量布尔函数中，只有 1920 个需要用到 12 条指令，其余那些所需指令数都比这个少。这 1920 个函数基本都大同小异，有的只是重新排列了自变量的位置，有的只是把某些自变量用其补值代替，而有的只是对整个函数结果取逻辑非而已。

2.23 实现 16 种二元布尔操作

某些 CPU 指令集支持全部 16 种二元布尔操作。这些指令中，很多都没有用。比如，$f(x,y)=0$ 这个函数，实际上就是清除某个寄存器的值而已，况且大多数计算机都有很多种方法可以清除一个寄存器的值，不一定非用布尔指令。然而，设计 CPU 架

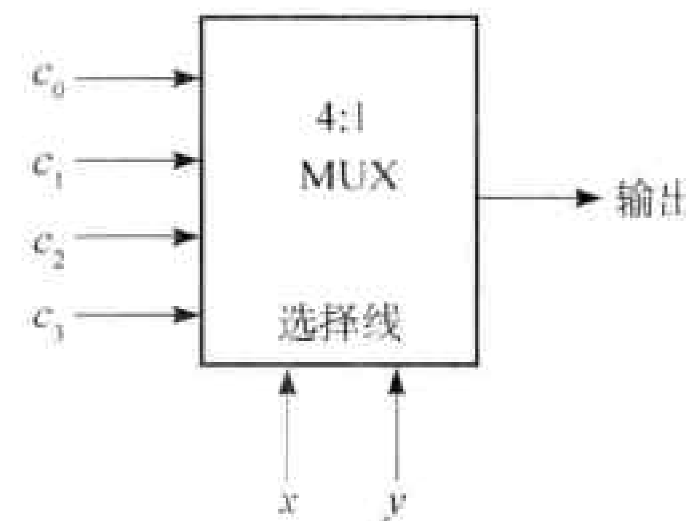
⊖ 推理的方法是，将实现变量数少 1 的布尔函数所需指令数乘以 2，再加 2。22=10×2+2。——译者注

构的人之所以要支持这16种布尔操作，其中一个原因就是有种简单且很常见的电路能够实现它们。

2.3节中的表2.1列出了16种二元布尔函数。为了将这些函数实现为指令，我们用每个函数的函数值来当指令码。表中每一栏从下至上的4个函数值，分别记为 c_0、c_1、c_2、c_3，将两个输入寄存器的值记为 x、y，则根据下列逻辑表达式即可构造一种能够实现16种二元布尔函数的电路来：

$$c_0xy+c_1x\bar{y}+c_2\bar{x}y+c_3\overline{xy}$$

例如当 $c_0=c_1=c_2=c_3=0$ 时，指令计算的就是零值函数 $f(x,y)=0$ 的值。如果 $c_0=1$ 而操作码的其余三个位元都是0，那就相当于实现了“逻辑与”指令，而如果 $c_0=c_3=0$ 且 $c_1=c_2=1$，那么上式的值也就是“逻辑异或”指令的操作结果。



可以用 n 个4：1 MUX数据选择器来实现此电路，其中 n 是计算机的字组长度。其中两条选择线（Select Line）分别表示 x 与 y，而每个MUX的数据输入端则代表操作码中4个二进制位。MUX是当前业界所用的一种标准部件，通常是一块运算速度很快的电路，其图示如下：

这块电路的功能是：根据 x 与 y 的值是00、01、10还是11在 c_0、c_1、c_2、c_3中选择一个值输出。它有点像那种带4个挡位的旋转开关（four-position rotary switch）。

上面这种电路很优雅，不过如果把16种操作码都用上，那么实现成本有些高。有很多种办法，只需8种操作码就可以实现出全部16种逻辑操作，不过代价是需要使用一些不太常规的逻辑操作。表2.3演示了其中一种方案。

表2.3　8种必备的逻辑指令

函数值	计算公式	指令助记符（指令名①，英文指令名）
0001	xy	and（逻辑与，and）
0010	$x\bar{y}$	andc（和另一个数的反码求逻辑与，and with complement）
0110	$x\oplus y$	xor（逻辑异或，exclusive or）
0111	$x+y$	or（逻辑或，or）
1110	$\overline{xy}$	nand（与非，negative and）
1101	$\overline{x\bar{y}}$或$\bar{x}+y$	cor（求补之后求逻辑或，complement and or）
1001	$\overline{x\oplus y}$或$x\equiv y$	eqv（逻辑等值，equivalence）
1000	$\overline{x+y}$	nor（或非，negative or）

① 表中andc又称“非蕴含”，cor又称“蕴含”，详情参见：http：//zh.wikipedia.org/wiki/逻辑运算符。andc运算通常用“and not”指令实现，该指令中文有时称“非与”，但为了避免同常见的“与非”操作混淆，译文多保留“and not”不译。——译者注

没有出现在表中的 8 种操作都可以用上述 8 种操作实现出来，具体做法是：交换 x 与 y 的输入值，或是把指令中两个表示寄存器的位段全部填成同一个寄存器[一]。详情参见本章末第 13 道习题。

IBM 的 POWER 架构 CPU 也用了这个方案，只是用“和另一个数的补值求逻辑或”（or with complement）取代了“求补之后求逻辑或”（complement and or）。表 2.3 所示的这个方案中，对前 4 条指令的执行结果分别求补，即可实现后 4 条指令。

布尔运算发展史

乔治·布尔[二]在 1854 年所著的《思维规律的研究》一书中，详细阐释了逻辑代数（the algebra of logic），然而它与今天大家熟知的“布尔代数”有某些区别。布尔本人用整数 1 和 0 来表示真（truth）与假（falsity），并演示了如何将生活中所说的“并且”、“或者”、“除了”等概念用普通数值代数（ordinary numerical algebra）规范地表达出来。此外，他还用普通代数规范地阐释了集合论中交集、分离集（disjoint sets）的并集、补集等概念。概率论中的概率都是 0 到 1 之间的实数，而布尔也用普通代数把这门理论加以规范。著作中还时常谈到一些哲学、宗教、法律等内容。

布尔被称为逻辑领域的大思想家，因为他把逻辑学规范化了，使得我们可以用普通的代数运算把一些复杂的逻辑命题忠实而精准地表达出来。

许多年之后，出现了一大批编程语言，它们都提供了 16 种布尔操作。IBM 的 PL/I 语言[三]（大约在 1966 年）引入了一个名叫 BOOL 的内置函数。在 BOOL（x,y,z）这个函数调用式中，z 是一个长度为 4 的位串（若该参数不是长度为 4 的位串，则此函数在必要时会将其转换为这种位串），而 x 与 y 则是两个长度相同的位串（如果这两个参数不是长度相同的位串，那么此函数在必要时会进行数据类型转换）。这里的 z 就代表函数要对 x 和 y 执行的布尔操作，例如二进制数 0000 表示恒零函数，0001 表示 xy 的值，而 0010 则表示 $x\bar{y}$ 的值，以此类推。

Wang System 2200B 计算机[四]中的 Basic 语言［Neum］也是这样，它（大约在 1974 年）也提供一种 BOOL 函数，不过，参与运算的不是位串或整数，而是字符串。

[一] 比如 and y，y 或 cor x，x。——译者注

[二] George Boole，1815—1864，爱尔兰数学家、哲学家。详情参见：http：//zh. wikipedia. org/wiki/乔治·布尔。——译者注

[三] PL/I，是 Programming Language One 的简写。当中的“I”其实是“一”的罗马数字。此语言运用于系统软件、图像、仿真、文字处理、网络、商业软件等领域，有些类似 PASCAL 语言。详情参见：http：//zh. wikipedia. org/wiki/PL/I。——译者注

[四] 是由王安电脑公司（Wang Laboratories，亦称王安实验室）推出的一款小型计算机（Minicomputer）。该公司是美籍华人计算机专家王安（1920—1990）创办的。详情参见：http：//en. wikipedia. org/wiki/Wang_2200 与 http：//en. wikipedia. org/wiki/Wang_Laboratories。——译者注

支持此类布尔操作的还有一门语言，就是后来改称 MacLisp 的 MIT PDP-6 Lisp[⊖][GLS1]。

2.24 习题

1. David de Kloet 提出了下列用于计算 snoob 函数[⊜]的代码。当 $x\neq 0$ 时，最终赋给 y 的值就是整个函数的结果：

$$y \leftarrow x+(x \,\&-x)$$
$$x \leftarrow x \,\&\neg y$$
$$\text{while}((x \,\&1)=0)\, x \leftarrow x \overset{u}{\gg} 1$$
$$x \leftarrow x \overset{u}{\gg} 1$$
$$y \leftarrow y \mid x$$

这个算法与 2.1.3 节中 Gosper 提到的那段代码基本相同，只是它没有用除法指令，而是通过 while 循环来实现右移的。由于除法指令通常比较耗时，所以假如 while 循环执行次数不多的话，那这个算法也许能和 Gosper 的算法一较高下。设 n 为位串 x、y 的长度，k 代表位串中值为 1 的位元个数，并假定我们把所有包含 k 个“1”的 x 值都用这种方法计算一遍，那么请问每次运用此算法时，while 循环的循环体平均执行几轮？

2. 本章正文中说过，如果左移操作的移位个数是个变量，那么它就不能按照“从右至左”的方式计算出来。然而有一个函数 [Knu8] $x \ll (x \,\&1)$，其左移操作的移位长度为变量，但是，使用下面这两个式子，却能把它用“从右至左”的方式算出来：

$$x+(x \,\&1) * x\text{，或}$$
$$x+(x \,\&(-(x \,\&1)))$$

请问这是怎么回事？你还能再想一个这种函数吗？

3. Dietz 指出，可以用下式计算两个无符号整数的均值。请证明此公式。

$$(x \,\&y)+((x \oplus y) \overset{u}{\gg} 1)$$

4. 想一个不会产生溢出的办法，来计算四个无符号整数的均值 $\lfloor (a+b+c+d)/4 \rfloor$。

5. 如果预知了 x 或 y 的第 31 位，那么就可以大大简化 2.12 节中的比较谓词了。假设 $y_{31}=0$，请在不使用比较指令的前提下，将原来计算表达式 $x \overset{u}{\leqslant} y$ 所用的 7 条指令简化为 3

⊖ MacLisp 也拼写为 MACLISP，是一种 Lisp 编程语言的方言。最早起源于 20 世纪 60 年代麻省理工学院的 MAC 计划（Project MAC），并因此得名。主要是理查德・格林布拉特（Richard Greenblatt）在 PDP-6 上研发的，之后由约翰・怀特（John L. White）负责维持与持续开发。从 20 世纪 70 年代开始，PDP-6 上的 Lisp 又发展出 BBN Lisp 等其他分支，为了与之区分，大家改称其为 Maclisp。详情参见：http://zh.wikipedia.org/wiki/Maclisp。而 PDP-6 则是 DEC 在 1963 年推出的一款计算机，详情参见：http://en.wikipedia.org/wiki/PDP-6。——译者注

⊜ 该函数用于求出下一个比 x 大，而值为 1 的位元个数又和 x 相同的数 y。——译者注

条基本 RISC 指令。

6. 假设有两个数，它们的值可能一样，也可能不一样。现在采用“首尾循环进位”法（end-around carry）将其相加，请证明：把第一次加法运算在最高位所产生的进位加到求和结果之后，不可能在最高位上产生第二次进位。
7. 假设负数用“反码”表示，那么请问如何以“首尾循环进位”方式实现加法操作？在这种计算方式下，由某个位置（不管哪个位置都行）上产生的进位，最多能传递几次？
8. 在一台只具备基本 RISC 指令集的计算机上（这种计算机不支持“和另一个数的补值求与”（and with complement）指令），用 3 条指令实现 MUX 操作：$(\boldsymbol{x}\ \&\boldsymbol{m})\mid(\boldsymbol{y}\ \&\sim\boldsymbol{m})$。
9. 用 4 条指令实现 $\boldsymbol{x}\oplus\boldsymbol{y}$。只准使用与、或、非三种逻辑操作。
10. 给定一个 32 位宽的字组 $\boldsymbol{x}$ 和两个存放在寄存器内的整数 $\boldsymbol{i}$、$\boldsymbol{j}$，编写一段代码，将 $\boldsymbol{x}$ 中序号为 $\boldsymbol{i}$ 的那个位元，复制到位置 $\boldsymbol{j}$ 上。$\boldsymbol{i}$、$\boldsymbol{j}$ 两个值之间没有联系，只要满足 $0\leqslant i,\ j\leqslant 31$ 就行。
11. 如果利用 2.22 节的定理把一个自变量个数为 n 的布尔函数逐层分解，那么请问需要多少条二元布尔指令才能将其实现出来？
12. 还有两种分解三变量布尔函数的办法，请证明它们：
    (a) $f(x,y,z)=g(x,y)\oplus\bar{z}h(x,y)$　（“负 Davio 分解”，negative Davio decomposition）
    (b) $f(x,y,z)=g(x,y)\oplus(z+h(x,y))$
13. 本章正文中说过，只要将两个输入值交换，或是让指令中表示两个寄存器的位段都指向同一个寄存器，那么就可以用表 2.3 中的 8 个指令实现出全部 16 种二元布尔操作。请说出具体算法。
14. 有 6 种二元布尔操作：$f(x,y)=0$，1，x，y，$\bar{x}$，$\bar{y}$，它们本质上都是常数函数或一元函数，现在假设在不考虑上面这几个函数的前提下，想设计一套只需一条指令即可实现剩余 10 种布尔操作的指令集，那么请问：有没有可能只用不到 8 种二元布尔指令（也就是表 2.3 中那种操作码）就能把这套指令集做出来？
15. 习题 13 已经证明：如果指令的两个操作数都是寄存器（R-R 指令，register-register instruction），那么仅需 8 种指令，就能实现全部 16 个二元布尔函数。现在请演示：如果指令的两个操作数一个是寄存器而另一个是常数（R-I 指令，register-immediate instruction），那么只用 6 种指令，就可以实现全部 16 个二元布尔函数。在使用 R-I 式指令时，不能交换两个操作数的位置，也不能填入两个相等的操作数，不过，可以对第二个输入值（也就是存放在指令码常量位段中的数 I）求补，或将其设定为其他值，这种操作不需要耗费执行时间。为了简洁起见，我们假设常量位段的宽度和通用寄存器所能容纳的位元数相等。
16. 请证明：并非所有三变量布尔函数都能用 3 条二元逻辑操作指令实现出来。

第 3 章

2 的幂边界

3.1 将数值上调/下调为 2 的已知次幂的倍数

如果想将无符号整数 x 下调为一个值最接近它同时还是 8 的倍数的数，那么只需执行 $x \,\&\, -8$ 就可以了。另外一种办法是 $(x \overset{u}{\gg} 3) \ll 3$。只要我们规定"下调"指的是向负无穷方向调整，那么上面两种算法就同样适用于带符号的整数了（例如 $(-37) \,\&\, (-8) = -40$）。

上调与下调一样容易。比方说，想将无符号整数 x 上调为值最接近它而又能被 8 整除的数，那么可以使用下列两式计算：

$$(x+7) \,\&\, -8 \text{ 或}$$
$$x+(-x \,\&\, 7)$$

只要我们规定"上调"的意思是向正无穷方向取整，那么这两个表达式就同样适用于带符号的整数。第 2 个表达式的第 2 项很有用，如果你想知道最少给 x 加上几才能把它变成 8 的倍数，那么用 $-x \,\&\, 7$ 就可以算出来［Gold］。

如果想把一个带符号整数朝 0 所在的方向调整为值最接近它且能被 8 整除的数，那么可以把上面两个表达式组合一下，这样就能得到下面这个算法了：

$$t \leftarrow (x \overset{s}{\gg} 31) \,\&\, 7;$$
$$(x+t) \,\&\, -8$$

上面这种算法的第 1 行也可以改成 $t \leftarrow (s \overset{s}{\gg} 2) \overset{u}{\gg} 29$。如果计算机没有和常量求与（and immediate）的指令，或是常量太大，没办法放在指令码的常量位段中，那么这个表达式就能派上用场了。

有些时候，舍入因子（rounding factor）不是对齐量，而是把对齐量[⊖]取以 2 为底的对数，然后用这个对数值来表示舍入因子（比方说，舍入因子值为 3，意思就是朝 8 的倍

⊖ alignment amount，也译作"调整量"、"调整边界"、"对齐边界"等，也就是舍入数值后的标准位置。舍入之后的数值必须是该值的整数倍。——译者注

数对齐[㊀])。在这种情况下，可以按照下列代码来舍入，其中 $k=\log_2$（对齐量）：

向下调整：　$x\ \&((-1)\ll k)$

　　　　　　$(x \overset{u}{\gg} k)\ll k$

向上调整：　$t\leftarrow(1\ll k)-1;(x+t)\ \&\neg t$

　　　　　　$t\leftarrow(-1)\ll k;(x-t-1)\ \&\ t$

3.2 调整到上一个/下一个 2 的幂

现在定义两个与“向下取整”和“向上取整”类似的函数，只是我们这次把舍入之后的结果规定为距离自变量最近的那个 2 的整数幂，而不是像原来那样调整到距其最近的整数。这两个函数的数学定义是：

$$\mathrm{flp2}(x)=\begin{cases}\text{未定义}, & x<0,\\ 0, & x=0,\\ 2^{\lfloor \log_2 x\rfloor}, & \text{其他};\end{cases}\qquad \mathrm{clp2}(x)=\begin{cases}\text{未定义}, & x<0,\\ 0, & x=0,\\ 2^{\lceil \log_2 x\rceil}, & \text{其他}.\end{cases}$$

函数名的头两个字母分别表示“floor”（向下取整）与“ceiling”（向上取整）。因此 flp2(x) 是小于等于 x 且最接近 x 的 2 的整数幂，而 clp2(x) 是大于等于 x 且最接近 x 的 2 的整数幂。即便 x 不是整数，也还是可以套用这两个函数的定义（例如，flp2(0.1)＝0.0625)[㊁]。这两个函数也满足一些与向上取整和向下取整函数类似的关系式。下面列出几个这样的式子，其中 n 为整数：

$\lfloor x\rfloor=\lceil x\rceil$	当且仅当 x 为整数	$\mathrm{flp2}(x)=\mathrm{clp2}(x)$	当且仅当 x 是 2 的幂或 0
$\lfloor x+n\rfloor=\lfloor x\rfloor+n$		$\mathrm{flp2}(2^n x)=2^n\,\mathrm{flp2}(x)$	
$\lceil x\rceil=-\lfloor -x\rfloor$		$\mathrm{clp2}(x)=1/\mathrm{flp2}(1/x)$，$x\neq 0$	

实际计算中我们只处理 x 为整数的情形，而且将其定为无符号数，这样的话，上面两个函数对所有的 x 值就都有定义了。此外，我们还规定，这些函数的值是将算数运算的正确结果与 2^{32} 取模后得到的（也就是说，如果 clp2($\boldsymbol{x}$) 中的 $\boldsymbol{x}$ 大于 $\mathbf{2^{31}}$，那么此函数的值就是 0)。下表列出了一些 $\boldsymbol{x}$ 的函数值。

$\boldsymbol{x}$	flp2($\boldsymbol{x}$)	clp2($\boldsymbol{x}$)
0	0	0
1	1	1
2	2	2
3	2	4
4	4	4

㊀ 因为 $\log_2 8=\log_2 2^3=3$。——译者注

㊁ 因为 $2^{-3}=0.125$，$2^{-4}=0.0625$，0.1 的值位于两者之间，所以对齐向下调整，就是 0.0625。——译者注

5	4	8
…	…	…
$2^{31}-1$	2^{30}	2^{31}
2^{31}	2^{31}	2^{31}
$2^{31}+1$	2^{31}	0
…	…	…
$2^{32}-1$	2^{31}	0

下面列出了 flp2 与 clp2 函数之间的关系式。在给定的条件下，可以利用这些关系式，根据其中一个函数的值，推算出另一个函数。

$$\begin{aligned}\text{clp2}(\boldsymbol{x}) &= 2\text{flp2}(\boldsymbol{x}-\mathbf{1}), & \boldsymbol{x}\neq 1,\\ &\text{flp2}(2\boldsymbol{x}-\mathbf{1}), & \mathbf{1}\leqslant\boldsymbol{x}\leqslant\mathbf{2}^{31},\\ \text{flp2}(\boldsymbol{x}) &= \text{clp2}(\boldsymbol{x}\overset{u}{\div}2+\mathbf{1}), & \boldsymbol{x}\neq\mathbf{0},\\ &= \text{clp2}(\boldsymbol{x}+\mathbf{1})\overset{u}{\div}2, & \boldsymbol{x}<\mathbf{2}^{31}.\end{aligned}$$

使用前导 0 计数指令（number of leading zeros instruction）很容易就能根据下列算式求出上调函数和下调函数的值。然而，如果要在 $\boldsymbol{x}=\mathbf{0}$ 及 $\boldsymbol{x}>\mathbf{2}^{31}$ 时使用这些关系式，那么就必须保证计算机的左移指令在移位量为 -1，32 及 63 时产生的结果是 $\mathbf{0}$。很多计算机（例如 Power PC）都采用“模 64”移位，这种移位方式符合上述要求。在移位量为 -1 的情况下，计算机朝相反方向移位也可以（也就是说，将左移 -1 位解读为右移 1 位也行）。

$$\begin{aligned}\text{flp2}(\boldsymbol{x}) &= \mathbf{1}\ll(\mathbf{31}-\text{nlz}(\boldsymbol{x}))\\ &= \mathbf{1}\ll(\text{nlz}(\boldsymbol{x})\oplus\mathbf{31})\\ &= \mathbf{0x8000\ 0000}\overset{u}{\gg}\text{nlz}(\boldsymbol{x})\\ \text{clp2}(\boldsymbol{x}) &= \mathbf{1}\ll(\mathbf{32}-\text{nlz}(\boldsymbol{x}-\mathbf{1}))\\ &= \mathbf{0x8000\ 0000}\overset{u}{\gg}(\text{nlz}(\boldsymbol{x}-\mathbf{1})-\mathbf{1})\end{aligned}$$

3.2.1 向下舍入

图 3.1 演示了如何在没有前导 0 计数指令时实现无分支的向下舍入算法。该算法的核心思路就是把最左方的“1”向右传播，执行起来共需 12 条指令。

图 3.2 分别采用两个简单的循环来计算同一个函数，其中所有变量都是无符号整数。在右侧的那个循环中，如果 $\boldsymbol{x}$ 不等于 $\mathbf{0}$，我们就反复“关闭” $\boldsymbol{x}$ 最右侧的“1”；如果 $\boldsymbol{x}$ 等于 $\mathbf{0}$，我们就把执行本轮循环前的 $\boldsymbol{x}$ 作为函数值返回。

左侧循环需要 $4\text{nlz}(\boldsymbol{x})+3$ 条指令实现，而如果忽略与 0 比较的指令，那么在 $\boldsymbol{x}\neq 0$ 时，右侧循环所需的指令数为 $4\text{pop}(\boldsymbol{x})$[⊖]。

⊖ pop($\boldsymbol{x}$) 就是 $\boldsymbol{x}$ 中值为 1 的位元个数。

```
unsigned flp2(unsigned x) {
   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >> 16);
   return x - (x >> 1);
}
```

图 3.1　用无分支代码将 x 下调为不大于 x 且值与之最近的 2 的幂

```
y = 0x80000000;             do {
while (y > x)                  y = x;
   y = y >> 1;                 x = x & (x - 1);
return y;                   } while(x != 0);
                            return y;
```

图 3.2　用简单的循环代码找出值不大于 x 且与之最接近的 2 的幂

3.2.2　向上舍入

利用刚才说的向右传播技巧，可以编写出一个巧妙的算法，把某个值向上调整为下一个 2 的幂。图 3.3 列出了此算法，代码不含分支，共需 12 条指令。

用刚才那种循环的办法实现向上舍入，效果并不好：

```
y = 1;
while (y < x)          //比较两个无符号数
   y = 2*y;
return y;
```

```
unsigned clp2(unsigned x) {
   x = x - 1;
   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >> 16);
   return x + 1;
}
```

图 3.3　寻找值不小于 x 且与之最接近的 2 的幂

上述代码在 $x=0$ 时的结果为 1，这恐怕与大家的预期不符[㊀]。而且一旦 $x\geqslant 2^{31}$，就会陷入死循环。在正常情况下，它执行时要花费 $4n+3$ 条指令，其中 n 是返回整数 y 所对应的 2 的幂指数[㊁]。因此，从执行所需的指令数来判断，在 $n\geqslant 3$（也就是 $x\geqslant 8$）时，该算法比无分支算法慢。

3.3　判断取值范围是否跨越了 2 的幂边界

假定内存被分隔成若干块（block），每一块的大小均为 2 的幂，且内存地址从 0 开始计数。这里说的“块”可以指字组、双字、页（page）等。现在给定起始地址 a 与长度 $l(l\geqslant 2)$，判断从 a 到 $a+l-1$ 之间的这一段内容是否跨越了两个块的边界。a 与 l 均为无符号数，其值不限，只要能容纳于寄存器中即可。

假如 $l=0$ 或 1，那么不论 a 为何值，都不会跨越块边界。而一旦 l 本身比块的大小还大，那么不论 a 为何值，都肯定会跨越块边界。假如 l 的值特别大（有可能大到 $a+l-1$ 的值已经突破了计算机容许的最大值，从而又折回到 0），那么就算内存范围的第一个字节与最后一个字节位于同一块内，也还是有可能跨越边界。

㊀ 按照 2.3 节中 clp2 函数的定义，当 x 为 0 时，函数值也应为 0，而不应为 1。——译者注

㊁ 例如 y 为 8 时，n 为 3，因为 $8=2^3$，也可以将 n 理解成 $\log_2 y$。——译者注

令人惊讶的是，在 IBM System/370 计算机[⊖]中，有一种非常精准的办法［CJS］能检测边界跨越。我们假设块大小为 4096 字节（也就是常见的内存页面大小），该方法代码如下：

```
O   RA,=A(-4096)
ALR RA,RL
BO  CROSSES
```

第 1 条指令的意思是将 RA（其中含有起始地址 a）与数字 0xFFFF FF00 取逻辑或。第 2 条指令则是将 RA 与长度 l 相加，并设置条件码。执行完这条逻辑加（add logical）指令后，如果发生进位，那么特征码的第 1 个位元置 1，如果 32 位寄存器 RA 的值非 0，则特征码的第 2 个位元置 1。若两个特征码位同时为 1，则最后一条指令会令程序转入分支。转入分支时，RA 的值就是该取值范围超过首个页面的那部分长度（这是一个未出现在需求中的附加功能）。

例如，若 $a=0$ 且 $l=4096$，那么会发生进位，然而结果寄存器中的值是 0。于是两个特征码不可能都是 1，所以程序也就不会转入 CROSSES 标签所指的分支了。

现在研究如何将上述算法移植到具备 RISC 指令集的计算机上，此类计算机通常没有那种“当发生进位且寄存器中运算结果非 0 时转入分支”的指令。为了描述起来方便，我们假定块的大小为 8 字节，这样的话，只有当发生进位（$(a \mid -8)+l \geqslant 2^{32}$）且寄存器结果非 0（$(a \mid -8)+l \neq 2^{32}$）时，［CJS］算法才会转入 CROSSES 分支。于是，上述算法也就等同于这个谓词的值：

$$(a \mid -8)+l > 2^{32}$$

而这个谓词的值，又与在计算 $((a \mid =8)-1)+l$ 的最后一个加法时的进位情况相同，所以在支持“进位时转入分支”（branch on carry）指令的计算机上，可以直接用此式来判断，这样的话，把载入常数 −8 的那条指令算上，一共需要 5 条。

如果计算机中没有“进位时转入分支”的指令，那么根据 2.13.3 节讲的“当且仅当 $\neg x \overset{u}{<} y$ 时，$x+y$ 才会发生进位”这一事实，可以得到下式：

$$\neg((\boldsymbol{a} \mid -\mathbf{8})-\mathbf{1}) \overset{u}{<} \boldsymbol{l}$$

使用 $\neg(\boldsymbol{x}-\mathbf{1})=-\boldsymbol{x}$ 等式子，可以推出下述几种等效的“边界跨越”谓词表达式：

$$-(\boldsymbol{a} \mid -\mathbf{8}) \overset{u}{<} \boldsymbol{l}$$

$$\neg(\boldsymbol{a} \mid -\mathbf{8})+\mathbf{1} \overset{u}{<} \boldsymbol{l}$$

$$(\neg \boldsymbol{a} \ \& \ \mathbf{7})+\mathbf{1} \overset{u}{<} \boldsymbol{l}$$

如果算上最后的分支指令，那么在大多数 RISC 架构的计算机中，上述算式需要 5 至 6 条指令。

⊖ IBM 公司 1970 年推出的大型电脑系列，是 IBM System/360 的后继产品。详情参见：https://en.wikipedia.org/wiki/IBM_System/370。——译者注

此问题还有另外一种解决思路。因为某个取值范围是否跨越 8 字节边界的充分必要条件是：

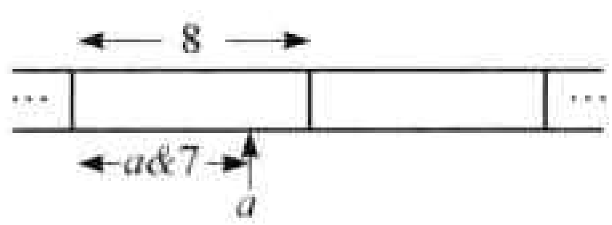
$$(a\ \&7)+l-1\geqslant 8$$

因为（当 l 很大时）可能会溢出，所以不能直接求此式的值。然而只要将其重排为 $\boldsymbol{8-(a\ \&7)<l}$，就能直接计算了（因为该式各部分均不会溢出）。于是就得到了这个表达式：

$$\boldsymbol{8-(a\ \&7)\overset{u}{<}l}$$

在绝大多数 RISC 架构的计算机中，这个式子都只需 5 条指令（如果有“立即数减法”指令，那就是 4 条）。若发生了跨越边界现象，那么可以用 $\boldsymbol{l-(8-(a\ \&7))}$ 求得超出第 1 块的长度。只需多用一条减法指令即可算出此值。

下边这张图［Kumar］直观地演示了此公式。其中 $\boldsymbol{a}$ &7 就是 $\boldsymbol{a}$ 在块中的偏移量，所以 $\boldsymbol{8-(a\ \&7)}$ 就是当前块中剩余的空间。



3.4　习题

1. 将无符号整数向 8 的倍数舍入。分别实现下列 3 种舍入标准：[㊀]
   (a) 三舍四入。[㊁]
   (b) 四舍五入。[㊂]
   (c) 三舍五入四成双[㊃]。在这种舍入方式下，一个除以 8 余 4 的数，应该舍入为距离其最近的 16 的倍数（这也叫“无偏差”舍入，“unbiased” rounding）。
2. 将无符号整数向 10 的倍数舍入。分别实现下列 3 种舍入标准。
   (a) 四舍五入。
   (b) 五舍六入。
   (c) 四舍六入五成双。在这种舍入方式下，一个除以 10 余 5 的数，应该舍入为距离其

㊀ 原文为 halfway case，也就是当待舍入的数与左右两个 8 的倍数之间的距离均为 4 时，是应该舍还是应该入。不同标准下的舍入结果有可能不同。比如待舍入的数是 20，它左边那个 8 的倍数是 16，右边那个 8 的倍数是 24，20 与两者的距离均为 4，那么 20 究竟舍为 16 还是入为 24，则取决于所选的标准。——译者注

㊁ 若该数除以 8 的余数大于等于 4，则向上调整，否则向下调整。——译者注

㊂ 若该数除以 8 的余数小于等于 4，则向下调整，否则向上调整。——译者注

㊃ 此处借用十进制下的俗语。若该整数除以 8 的余数小于 4，则向下调整，大于 4 则向上调整。若恰好为 4，且商为偶数，则向下调整；若商为奇数，则向上调整，调整后的值必定是 8 的双数倍。比如 28 这个数，与 24 和 32 的距离都是 4，然而 28 与 8 的商为 3，是奇数，所以应该上调为 32，而不能下调为 24，因为 24 不是 8 的偶数倍。——译者注

最近的 10 的偶数倍。[1]

此题可任意使用除法、求余、乘法指令，而且不必考虑那些与最大的无符号整数非常接近的值。

3. 用 C 语言编写一个实现“不对齐加载”（unaligned load）功能的函数。该函数接收一个起始地址 a，然后把从 a 到 $a+3$ 的 4 个字节视为一个整数，载入 32 位寄存器中。起始地址参数 a 所指的内容，是待加载整数中权重最低的那个字节（也就是说，假定计算机是以“小端序”[2]的方式存放多字节整数的）。函数代码不要有分支语句，而且最多只能执行两次加载指令。如果 a 本身恰好位于字组边界处，则函数不应读取 $a+4$ 这个地址中的内容，因为此字节可能处于一个“读保护块”（read-protected block）中。

---

[1] 即按照“四舍六入五成双”标准舍入之后的数必定是 20 的倍数。其中“成双”一词的意思是，如果发现待舍入的值除以 10 的余数为 5，那么一定要把它调整成 10 的双数倍（也就是 20 的倍数）。例如 35 与 30 和 40 的距离均为 5，而 35 与 10 的商为 3，是奇数，所以应该上调为 40，而不能下调为 30，因为 30 不是 10 的双数倍。详情参见：http：//zh. wikipedia. org/wiki/数值简化规则。——译者注

[2] little-endian，又称“小尾序”，它会把权重最低的字节存放在内存地址最小处。详情参见：http：//zh. wikipedia. org/wiki/字节序。——译者注

第 4 章
算术边界

4.1 检测整数边界

“边界检测”就是判断整数 x 是否处于由 a、b 两数所界定的范围内。换言之，就是判断下面这个式子是否成立：

$$a \leqslant x \leqslant b$$

首先假定所有数字都是带符号整数。

此技术的一个重要用途就是检测数组下标。比方说，有一个一维数组 A，其有效下标为 1 至 10。现在编译器为了判断 $A(i)$ 这个数组元素引用是否合法，可能会生成代码来检测下式是否成立：

$$1 \leqslant i \leqslant 10$$

如果 i 的值不满足该式，那么程序可能会转入分支或是触发中断。本节我们要告诉大家的就是，可以用一条简单的比较指令［PL8］来完成与上式等效的判断：

$$i-1 \overset{u}{\leqslant} 9$$

按照这个式子来编码效果可能更好，因为它只需要 1 条比较分支（compare-branch）指令或比较中断（compare-trap）指令，而且在计算数组元素地址时，很可能会用到 $i-1$ 的值。

那么，下面这种实现方式

$$a \leqslant x \leqslant b \Rightarrow \boldsymbol{x}-\boldsymbol{a} \overset{u}{\leqslant} \boldsymbol{b}-\boldsymbol{a}$$

是不是总能求出正确结果呢？在减法发生溢出时还管用吗？答案是肯定的：只要保证 $a \leqslant b$，那么这种方法就总是能求出正确结果。在执行数组边界检测时，编程语言的规则可能会要求数组中值为 0 或负的元素总数（或是在任意轴上的元素总数）不得超过一定的量。这条规则的遵守情况可以在编译时检测，也可以在分配数组时动态检测。在这种情况下，上面那个变形方式就是正确的，接下来我们将给出证明。

下面这条引理（LEMMA）有助于证明，而且这条引理本身也有其价值。

引理 若 a 与 b 均为带符号整数，且 $a \leqslant b$，那么如果将 $\boldsymbol{b}-\boldsymbol{a}$ 的结果解读为无符号整

数，则其值就是算术运算 $b-a$ 的正确结果。

证明（假设电脑字组长度为32）：因为 $a \leqslant b$，所以 $b-a$ 的正确结果的取值范围必定在0至 $(2^{31}-1)-(-2^{31})$ 即 $2^{32}-1$ 之间。如果两数之差位于0至 $2^{31}-1$ 之间，则计算机给出的结果就是正确结果（因为这一范围内的运算结果可以表示为带符号整数），而且其符号位为0。因此，在这种情况下，无论将其视为带符号数或是无符号数，结果都是正确的。

如果两数之差的正确结果位于 2^{31} 至 $2^{32}-1$ 之间，那么计算机算出来的结果与真正结果之间的差距肯定为 2^{32} 的倍数（因为此时结果无法用带符号数表示）。于是，这样算出来的结果（以带符号数表示）肯定位于 -2^{31} 至 -1 之间。计算机求出的结果比真正结果少了 2^{32}，且符号位为1。而将这个结果解读为无符号数，就等于在它上面加了 2^{32}，因为在无符号数中，符号位的权重是 $+2^{31}$，而不是 -2^{31}。因此，在这种情况下，结果也是正确的。 ■

证明完上述引理后，来看"边界检测定理"（bound theorem）：

定理 若 a 与 b 为带符号整数，且 $a \leqslant b$，那么

$$a \leqslant x \leqslant b = \boldsymbol{x}-\boldsymbol{a} \overset{u}{\leqslant} \boldsymbol{b}-\boldsymbol{a} \tag{1}$$

证明：根据 x 的值分3种情况讨论。不管那种情况，根据前述引理，因为 $a \leqslant b$，所以只要将（1）式中 $\boldsymbol{b}-\boldsymbol{a}$ 的值解读为无符号数，其值就必定等于 $b-a$ 的算术结果。

第1种情况：$x<a$。在这种情况下，将 $\boldsymbol{x}-\boldsymbol{a}$ 视为无符号数，其值就是 $x-a+2^{32}$，而不论 x 与 b 为何值（只要在32位二进制整数范围内），二者必有下述关系：

$$x+2^{32}>b$$

因此

$$x-a+2^{32}>b-a$$

刚才已经由引理推知，$x-a+2^{32}$ 就是将 $\boldsymbol{x}-\boldsymbol{a}$ 解读为无符号数时的值，所以：

$$\boldsymbol{x}-\boldsymbol{a} \overset{u}{>} \boldsymbol{b}-\boldsymbol{a}$$

在这种情况下，等式（1）的两端均为**假**。

第2种情况：$a \leqslant x \leqslant b$。此时可得出这个算术不等式：$x-a \leqslant b-a$。由于 $a \leqslant x$，根据引理，如果将 $\boldsymbol{x}-\boldsymbol{a}$ 解读为无符号数，那么其值就是 $x-a$ 的算术结果。因此：

$$\boldsymbol{x}-\boldsymbol{a} \overset{u}{\leqslant} \boldsymbol{b}-\boldsymbol{a}$$

在这种情况下，等式（1）的两端均为**真**。

第3种情况：$x>b$。此时可得算术不等式：$x-a>b-a$。因为 $x>b$，且 $b \geqslant a$，所以 $x>a$。根据引理，若将 $\boldsymbol{x}-\boldsymbol{a}$ 视为无符号数，那么其值就是 $x-a$ 的算术结果。故而：

$$\boldsymbol{x}-\boldsymbol{a} \overset{u}{>} \boldsymbol{b}-\boldsymbol{a}$$

在这种情况下，等式（1）两端均为**假**。

上述定理在 a、b 均为无符号整数时也成立。这是因为：如果两数均为无符号数，那

么引理显然成立，而上面分 3 种情况讨论的证明过程也成立，故整个定理成立。 ■

下面这几个彼此相似的变形公式与上述定义一样，都可以做边界检测。不管是将 a、b、x 解读为带符号数还是无符号数，这些公式都成立。

$$
\begin{aligned}
&\text{若 } a \leqslant b, \text{则 } a \leqslant x \leqslant b = \boldsymbol{x}-\boldsymbol{a} \overset{u}{\geqslant} \boldsymbol{b}-\boldsymbol{a}=\boldsymbol{b}-\boldsymbol{x} \overset{u}{\leqslant} \boldsymbol{b}-\boldsymbol{a} \\
&\text{若 } a \leqslant b, \text{则 } a \leqslant x < b = \boldsymbol{x}-\boldsymbol{a} \overset{u}{<} \boldsymbol{b}-\boldsymbol{a} \\
&\text{若 } a \leqslant b, \text{则 } a < x \leqslant b = \boldsymbol{b}-\boldsymbol{x} \overset{u}{<} \boldsymbol{b}-\boldsymbol{a} \\
&\text{若 } a < b, \text{则 } a < x < b = \boldsymbol{x}-\boldsymbol{a}-\mathbf{1} \overset{u}{<} \boldsymbol{b}-\boldsymbol{a}-\mathbf{1}=\boldsymbol{b}-\boldsymbol{x}-\mathbf{1} \overset{u}{<} \boldsymbol{b}-\boldsymbol{a}-\mathbf{1}
\end{aligned}
\tag{2}
$$

最后一条规则中，$\boldsymbol{b}-\boldsymbol{a}-\mathbf{1}$ 也可以写为 $\boldsymbol{b}+\neg\boldsymbol{a}$。

还有一些与上面各式差别较大的变换公式，可以用来进行 $-2^{n-1} \leqslant x \leqslant 2^{n-1}-1$ 形式的检测。这种检测可以判断带符号整数 x 是否能够用 n 个二进制位的补码正确表示出来。比方说，在 $n=8$ 时，下面这些检测式是等价的：

a. $\mathbf{-128} \leqslant \boldsymbol{x} \leqslant \mathbf{127}$

b. $\boldsymbol{x}+\mathbf{128} \overset{u}{\leqslant} \mathbf{255}$

c. $(\boldsymbol{x} \overset{s}{\gg} 7)+\mathbf{1} \overset{u}{\leqslant} \mathbf{1}$

d. $\boldsymbol{x} \overset{s}{\gg} 7=\boldsymbol{x} \overset{s}{\gg} \mathbf{31}$

e. $(\boldsymbol{x} \overset{s}{\gg} 7)+(\boldsymbol{x} \overset{u}{\gg} 31)=\mathbf{0}$

f. $(\boldsymbol{x} \ll 24) \overset{s}{\gg} 24=\boldsymbol{x}$

g. $\boldsymbol{x} \oplus (\boldsymbol{x} \overset{s}{\gg} 31) \leqslant \mathbf{127}$

运用本节刚讲过的内容，很容易就可以推出等式（b）。等式（c）也一样，只是先要把 $\boldsymbol{x}$ 带符号右移 7 位，然后再运用上述定理。只有（a）式与（b）式中的常数无法容纳在计算机的比较指令或加法指令内的常数位段时，才值得考虑使用等式（c）至（f）（可能也包括（g））。

还有一种特殊情况也与 2 的幂有关：

$$0 \leqslant x \leqslant 2^n-1 \Leftrightarrow (\boldsymbol{x} \overset{u}{\gg} n)=\mathbf{0}$$

或者写成更通用的形式：

$$a \leqslant x \leqslant a+2^n-1 \Leftrightarrow ((\boldsymbol{x}-\boldsymbol{a}) \overset{u}{\gg} n)=\mathbf{0}$$

4.2 通过加减法传播边界

某些优化编译器会对表达式执行“取值范围分析”（Range Analysis）。在此过程中，编译器要分析程序中出现的每个表达式，并确定其取值范围的上界与下界。尽管这种优

化实际上提升不了多少性能，但是它还是可以促进一些代码改进技术。比如省略C语言“switch”语句中的范围检测，以及省略对编译器为了调试而生成的下标所做的范围检测。

假设两个变量 x、y 的取值范围如下（其中各数均无符号）：

$$\begin{aligned} &a \leqslant x \leqslant b，且 \\ &c \leqslant y \leqslant d \end{aligned} \tag{3}$$

那么如何算出 $\boldsymbol{x}+\boldsymbol{y}$、$\boldsymbol{x}-\boldsymbol{y}$、$-\boldsymbol{x}$ 的“紧边界”[⊖]呢？当然可以用算术式 $a+c \leqslant x+y \leqslant b+d$ 来求 $\boldsymbol{x}+\boldsymbol{y}$ 的取值范围，然而问题是其中的加法会溢出。

下列定理可以算出上述三者的取值范围。

定理 若 a、b、c、d、x、y 均为无符号整数，且

$$\boldsymbol{a} \overset{u}{\leqslant} \boldsymbol{x} \overset{u}{\leqslant} \boldsymbol{b},$$

$$\boldsymbol{c} \overset{u}{\leqslant} \boldsymbol{y} \overset{u}{\leqslant} \boldsymbol{d}$$

那么

$$\begin{aligned} &若\ a+c \leqslant 2^{32}-1，且\ b+d \geqslant 2^{32}，则\ \boldsymbol{0} \overset{u}{\leqslant} \boldsymbol{x}+\boldsymbol{y} \overset{u}{\leqslant} \boldsymbol{2^{32}-1}； \\ &否则\ \boldsymbol{a}+\boldsymbol{c} \overset{u}{\leqslant} \boldsymbol{x}+\boldsymbol{y} \overset{u}{\leqslant} \boldsymbol{b}+\boldsymbol{d}； \end{aligned} \tag{4}$$

$$\begin{aligned} &若\ a-d<0，b-c \geqslant 0，则\ \boldsymbol{0} \overset{u}{\leqslant} \boldsymbol{x}-\boldsymbol{y} \overset{u}{\leqslant} \boldsymbol{2^{32}-1}； \\ &否则\ \boldsymbol{a}-\boldsymbol{d} \overset{u}{\leqslant} \boldsymbol{x}-\boldsymbol{y} \overset{u}{\leqslant} \boldsymbol{b}-\boldsymbol{c} \end{aligned} \tag{5}$$

$$\begin{aligned} &若\ a=0，b \neq 0，则\ \boldsymbol{0} \overset{u}{\leqslant} -\boldsymbol{x} \overset{u}{\leqslant} \boldsymbol{2^{32}-1}； \\ &否则\ -\boldsymbol{b} \overset{u}{\leqslant} -\boldsymbol{x} \overset{u}{\leqslant} -\boldsymbol{a} \end{aligned} \tag{6}$$

不等式（4）的意思是说，$\boldsymbol{x}+\boldsymbol{y}$ 的取值范围“一般来讲”就在 $\boldsymbol{a}+\boldsymbol{c}$ 与 $\boldsymbol{b}+\boldsymbol{d}$ 之间，但是如果 $\boldsymbol{a}+\boldsymbol{c}$ 不溢出而 $\boldsymbol{b}+\boldsymbol{d}$ 溢出，那么取值范围就在0与最大无符号整数之间。不等式（5）也可类似解读，只是在 $\boldsymbol{a}-\boldsymbol{d}$ 的算术结果小于0时，计算机在计算 $\boldsymbol{a}-\boldsymbol{d}$ 时会产生负向溢出。

证明：若 $\boldsymbol{a}+\boldsymbol{c}$ 与 $\boldsymbol{b}+\boldsymbol{d}$ 均不溢出，则对给定范围内的两值 $\boldsymbol{x}$、$\boldsymbol{y}$，其和 $\boldsymbol{x}+\boldsymbol{y}$ 也不可能溢出，这样的话，计算机算出来的结果与算术结果一致，于是不等式（4）的第2部分成立。若 $\boldsymbol{a}+\boldsymbol{c}$ 与 $\boldsymbol{b}+\boldsymbol{d}$ 均溢出，则 $\boldsymbol{x}+\boldsymbol{y}$ 也必定溢出。此时按照算术运算规则，显然可知：

$$a+c-2^{32} \leqslant x+y-2^{32} \leqslant b+d-2^{32}$$

当 $\boldsymbol{a}+\boldsymbol{c}$、$\boldsymbol{b}+\boldsymbol{d}$、$\boldsymbol{x}+\boldsymbol{y}$ 三者均溢出时，溢出后的值也就是上述不等式中的那三项，因此，在这种情况下，下式依然成立：

⊖ tight bound，也就是由范围尽可能小的两个极值所定义的边界。例如在 $5 \leqslant x \leqslant 10$，$6 \leqslant y \leqslant 12$ 的情况下，将两式相加，可得 $11 \leqslant x+y \leqslant 22$。此时当然可以用 $0 \leqslant x+y \leqslant 50$，$-100 \leqslant x+y \leqslant 200$ 等较为宽泛的不等式来界定 $x+y$ 的取值范围，然而，由11和22这两个边界值所界定的取值范围是所有说法中跨度最小的一个（前两者的跨度分别为50和300，而后者仅为11），故而称为“紧”边界或“紧”界。本书译文将酌情使用“最窄取值范围”等俗语来称呼这一概念。——译者注

$$\boldsymbol{a+c \overset{u}{\leqslant} x+y \overset{u}{\leqslant} b+d}$$

若 $\boldsymbol{a+c}$ 不溢出而 $\boldsymbol{b+d}$ 溢出，那么：

$$a+c \leqslant 2^{32}-1 \text{ 且 } b+d \geqslant 2^{32}$$

因为 $x+y$ 会取遍从 $a+c$ 到 $b+d$ 这个范围内的每一个值，所以它也会等于 $2^{32}-1$ 与 2^{32} 这两个值。也就是说，计算机算出来的 $\boldsymbol{x+y}$，结果有可能是 $2^{32}-1$ 与 0（虽然 $\boldsymbol{x+y}$ 未必取遍该范围内的每一个值，但确实能够取到这两个值）。

最后一种情况是不可能发生的，因为 $a \leqslant b$ 且 $c \leqslant d$，所以绝对不可能出现 $\boldsymbol{a+c}$ 溢出而 $\boldsymbol{b+d}$ 却不溢出的情况。

这样就证明了不等式（4），不等式（5）的证明与之类似，只不过“溢出”一词指的是两数的算术差小于 0。

将不等式（5）中的 $\boldsymbol{a}$、$\boldsymbol{b}$ 取 0，然后修改一下变量名，就可以证明不等式（6）了。（如果 $\boldsymbol{x}$ 是无符号数，那么表达式 $-\boldsymbol{x}$ 在计算机中的值可以用 $2^{32}-\boldsymbol{x}$ 或 $\neg \boldsymbol{x}+\mathbf{1}$ 算出来。） ■

由于无符号数的溢出很好判断（参见 2.13.3 节），所以如图 4.1 所示，很容易就能把上述不等式所对应的加减法取值范围用代码表示出来，其中 s 与 t 分别表示取值范围的下限及上限。

带符号数加减法的取值范围

带符号数加减法的取值范围则不那么明确。我们还是像前面那样假定变量 x 与 y 的取值范围如下（其中各数均带符号）：

$$a \leqslant x \leqslant b \text{ 且}$$
$$c \leqslant y \leqslant d$$

```
s = a + c;
t = b + d;
if (s >= a && t < b) {
   s = 0;
   t = 0xFFFFFFFF;}
```

```
s = a - d;
t = b - c;
if (s > a && t <= b) {
   s = 0;
   t = 0xFFFFFFFF;}
```

图 4.1 求两个无符号数进行加减法运算之后的取值范围⊖

我们想求出 $\boldsymbol{x+y}$、$\boldsymbol{x-y}$、$-\boldsymbol{x}$ 的紧边界。采用与无符号数非常相似的论证过程，可以得到下列各式，它们都描述了两个带符号数相加之后的取值范围。

$$\begin{array}{ll}
a+c<-2^{31}, b+d<-2^{31}: & \boldsymbol{a+c \leqslant x+y \leqslant b+d} \\
a+c<-2^{31}, b+d \geqslant -2^{31}: & \boldsymbol{-2^{31} \leqslant x+y \leqslant 2^{31}-1} \\
-2^{31} \leqslant a+c<2^{31}, b+d<2^{31}: & \boldsymbol{a+c \leqslant x+y \leqslant b+d} \\
-2^{31} \leqslant a+c<2^{31}, b+d \geqslant 2^{31}: & \boldsymbol{-2^{31} \leqslant x+y \leqslant 2^{31}-1} \\
a+c \geqslant 2^{31}, b+d \geqslant 2^{31}: & \boldsymbol{a+c \leqslant x+y \leqslant b+d}
\end{array} \tag{7}$$

⊖ 此处原文为“Propagating unsigned bounds through addition and subtraction operations”，直译就是“通过加减法操作传播无符号数的边界”，译文中换用通俗说法转述之。——译者注

第1行的意思是，如果 $a+c$ 与 $b+d$ 都朝负向溢出[⊖]，那么计算机算出的 $\boldsymbol{x}+\boldsymbol{y}$ 就一定位于它所算出的 $\boldsymbol{a}+\boldsymbol{c}$ 和 $\boldsymbol{b}+\boldsymbol{d}$ 这两个值之间。这是因为三者都要比实际算数运算的结果大 2^{32}。第2行的意思是，如果 $a+c$ 朝负方向溢出，而 $b+d$ 要么不溢出，要么朝正方向溢出[⊜]，那么不难证明：计算机算出来的 $\boldsymbol{x}+\boldsymbol{y}$ 就一定位于最小的负数与最大的正数之间（尽管未必能够取到其中每一个值）。其余各行亦可做类似解读。

把 y 的取值范围按照下述方式重写，即可推出带符号数减法的取值范围：

$$-d \leqslant -y \leqslant -c$$

将上式与描述加法取值范围的那5个式子分别相加，即可得出下列各式：

$$\begin{aligned}
&a-d<-2^{31}, b-c<-2^{31}: \boldsymbol{a}-\boldsymbol{d} \leqslant \boldsymbol{x}-\boldsymbol{y} \leqslant \boldsymbol{b}-\boldsymbol{c} \\
&a-d<-2^{31}, b-c \geqslant -2^{31}: -\mathbf{2}^{\mathbf{31}} \leqslant \boldsymbol{x}-\boldsymbol{y} \leqslant \mathbf{2}^{\mathbf{31}}-\mathbf{1} \\
&-2^{31} \leqslant a-d<2^{31}, b-c<2^{31}: \boldsymbol{a}-\boldsymbol{d} \leqslant \boldsymbol{x}-\boldsymbol{y} \leqslant \boldsymbol{b}-\boldsymbol{c} \\
&-2^{31} \leqslant a-d<2^{31}, b-c \geqslant 2^{31}: -\mathbf{2}^{\mathbf{31}} \leqslant \boldsymbol{x}-\boldsymbol{y} \leqslant \mathbf{2}^{\mathbf{31}}-\mathbf{1} \\
&a-d \geqslant 2^{31}, b-c \geqslant 2^{31}: \boldsymbol{a}-\boldsymbol{d} \leqslant \boldsymbol{x}-\boldsymbol{y} \leqslant \boldsymbol{b}-\boldsymbol{c}
\end{aligned}$$

将 $a=b=0$ 代入上述各式，然后略去不可能出现的取值组合，最后化简并重新调整变量名，即可得出下面这组描述相反数的取值范围的不等式：

$$\begin{aligned}
&a=-2^{31}, b=-2^{31}: -\boldsymbol{x}=-\mathbf{2}^{\mathbf{31}} \\
&a=-2^{31}, b \neq -2^{31}: -\mathbf{2}^{\mathbf{31}} \leqslant -\boldsymbol{x} \leqslant \mathbf{2}^{\mathbf{31}}-\mathbf{1} \\
&a \neq -2^{31}: -\boldsymbol{b} \leqslant -\boldsymbol{x} \leqslant -\boldsymbol{a}
\end{aligned}$$

用C语言求带符号数的运算结果的取值范围，代码写起来有点乱。我们此处只谈加法。看上去最简单的办法，就是判断待检测的两个变量是否符合（7）中运算结果为负极值到正极值之间的那两种情况。若两个操作数均为负，而结果却为正，那么就会发生负方向溢出（参见2.13.1节）。因此，为了判断是否会出现 $a+c<-2^{31}$ 的情况，我们可令 s=a+c，然后用“if(a<0 && c<0 & & s>=0) ...”这样的代码来检测。然而如果想求得更高效[⊜]的代码，那么可以直接在算式中的变量上进行逻辑操作，这样的话，操作结果的符号位也就可以表示逻辑操作的真/假了。于是，可以把上面那个 if 语句改写为“if ((a& c& ~s)<0) ...”。考虑到上述这些因素后，就有了图4.2中大家看到的这段代码。

```
s = a + c;
t = b + d;
u = a & c & ~s & ~(b & d & ~t);
v = ((a ^ c) | ~(a ^ s)) & (~b & ~d & t);
if ((u | v) < 0) {
   s = 0x80000000;
   t = 0x7FFFFFFF;}
```

图4.2 求两个带符号数相加之后的取值范围

如果 a+c 发生负方向溢出，而 b+d 不发生负方向溢出，那么 u 的值就是**真**（也就是其符号位为1）。如果 a+c 不溢出而 b+d 朝正方向溢出，那么 v 的值就是**真**。前面说的“a+c

[⊖] overflow in the negative direction，意思是两个负数之和太小，从而无法正确存放于运行结果之中，故曰“负方向溢出”。——译者注

[⊜] overflow in the positive direction，意思是两个正数之和太大，从而无法正确存放于运行结果之中，故曰“正方向溢出”。——译者注

[⊜] 这里所谓的“高效”，是从代码量精简、分支数少的角度考虑的。如果从提升代码运行速度的角度看，那么在两数之和的取值边界不会太大时，先检测未溢出的情况会更快些。

不溢出”，也可以表述为“a 和 c 的正负号不同，或 a 和 s 的正负号相同”。代码中的“if”检测语句与“if(u<0 || v<0)”等效，也就是判断 u 和 v 中是不是至少有一个值为真。

4.3 通过逻辑操作传播边界

与前一节相同，我们还是假定两个变量 x 与 y 的取值范围如下（其中所有值均为无符号数）：

$$
\begin{aligned}
a \leqslant x \leqslant b \\
c \leqslant y \leqslant d
\end{aligned} \tag{8}
$$

那么，$\boldsymbol{x} \mid \boldsymbol{y}$、$\boldsymbol{x} \,\&\, \boldsymbol{y}$、$\boldsymbol{x} \oplus \boldsymbol{y}$、$\neg \boldsymbol{x}$ 的紧边界分别是什么？

将不等式（8）中的各式与 2.3 节中的那些不等式结合起来，再运用 $\neg \boldsymbol{x} = \mathbf{2}^{32} - \mathbf{1} - \boldsymbol{x}$ 这一等式，即可得出下列各式：

$$
\begin{aligned}
\max(\boldsymbol{a}, \boldsymbol{c}) \leqslant (\boldsymbol{x} \mid \boldsymbol{y}) &\leqslant \boldsymbol{b} + \boldsymbol{d} \\
\mathbf{0} \leqslant (\boldsymbol{x} \,\&\, \boldsymbol{y}) &\leqslant \min(\boldsymbol{b}, \boldsymbol{d}) \\
\mathbf{0} \leqslant (\boldsymbol{x} \oplus \boldsymbol{y}) &\leqslant \boldsymbol{b} + \boldsymbol{d} \\
\neg \boldsymbol{b} \leqslant \neg \boldsymbol{x} &\leqslant \neg \boldsymbol{a}
\end{aligned}
$$

上面各式要假定 $\boldsymbol{b} + \boldsymbol{d}$ 不会发生溢出。这些式子计算起来很容易，而且编译器也可以用它们来实现与上节中提到的那两项代码优化技术相似的一些功能。然而，在上面这组式子中，前两个不等式给出的并不是“紧边界”。举例来说，假设有如下两个二进制数的不等式：

$$
\begin{aligned}
\mathbf{00010} \leqslant \boldsymbol{x} \leqslant \mathbf{00100} \\
\mathbf{01001} \leqslant \boldsymbol{y} \leqslant \mathbf{10100}
\end{aligned} \tag{9}
$$

那么，通过逐一尝试 $\boldsymbol{x}$ 与 $\boldsymbol{y}$ 取值的 36 种组合[⊖]，我们可以得出：$\mathbf{01010} \leqslant (\boldsymbol{x} \mid \boldsymbol{y}) \leqslant \mathbf{10111}$。这个不等式的左边界既不是 $\max(\boldsymbol{a}, \boldsymbol{c})$，也不是 $\boldsymbol{a} \mid \boldsymbol{c}$，而其右边界既不是 $\boldsymbol{b} + \boldsymbol{d}$，也不是 $\boldsymbol{b} \mid \boldsymbol{d}$。

所以还是要思考，根据不等式（8）中给出的 a、b、c、d，如何求出各种逻辑操作的最窄取值范围呢？首先考虑 $\boldsymbol{x} \mid \boldsymbol{y}$ 最小取值。大家有理由猜测：当 $\boldsymbol{x}$ 与 $\boldsymbol{y}$ 都位于其取值范围的左边界时，$\boldsymbol{x} \mid \boldsymbol{y}$ 的值最小，此时该值应该是 $\boldsymbol{a} \mid \boldsymbol{c}$。然而从范例（9）中可知，真正的最小值比这个还要小。

想要找出最小值，我们的办法是从 $\boldsymbol{x} = \boldsymbol{a}$ 且 $\boldsymbol{y} = \boldsymbol{c}$ 开始，逐步寻找某个增量，给 $\boldsymbol{x}$ 或 $\boldsymbol{y}$ 其中之一加上此增量，然后计算 $\boldsymbol{x} \mid \boldsymbol{y}$，看看这个增量能不能降低 $\boldsymbol{x} \mid \boldsymbol{y}$ 的取值。如果能，那么 $\boldsymbol{x} \mid \boldsymbol{y}$ 的左边界就是这个降低之后的值。实际操作时，我们并不把 $\boldsymbol{a}$、$\boldsymbol{c}$ 赋值给 $\boldsymbol{x}$、$\boldsymbol{y}$，而是直接用 $\boldsymbol{a}$、$\boldsymbol{c}$ 来试验：让其中一个值变大，然后判断新的取值范围是否成立，同时检

⊖ $\boldsymbol{x}$ 的取值范围是 [2, 4]，而 $\boldsymbol{y}$ 的取值范围是 [9, 20]，故取值组合的种类数为：(4－2＋1)＊(20－9＋19)＝3＊12＝36。——译者注

测新的值能不能降低 $a \mid c$ 的取值。

计算过程是，从左至右扫描 a、c 两数的每一个位元。如果两个位元均为 0，那么结果的这一位也是 0。如果二者皆为 1，那么结果的这一位也是 1（在上述两种情况下，显然无法通过增加 x 或 y 的值来让 $x \mid y$ 的结果变小）。如果遇到上述两种情况，那么就继续扫描下一位。要是在扫描两数的过程中发现，其中有一个数在某个位置上的位元值是 1，而另一个数在该位置的位元值是 0，那么就试着把那个“0”变成“1”，并且把该位置右侧的位元全设为“0”，以此降低 $a \mid c$ 的值。这么做绝对不可能提升 $a \mid c$ 的值，因为 a、c 之中值未改变的那个数在这一位置上的位元值已经是“1”了，所以在 $a \mid c$ 的结果中，这一位也必然是“1”。此时我们先把待试验的那个数字在这一位置上的“0”改成“1”，然后把其右侧的位元全置为“0”。如果这个值小于等于相应不等式中的上限，那么就说明这一改动成立，于是，我们就把修改之后的值同 a、c 之中值未改变的那个数取或运算。若是修改之后的值超出了相应不等式中的上限，那么就说明这一改动不成立，继续扫描下一位即可。

整个过程就是这样。修改了 a、c 之中的某个数后，看上去似乎还应该继续往下扫描，试试还有没有可能进一步降低 $a \mid c$ 的值。然而实际上却不用再往下找了，就算能找到一个可以把 0 改成 1 的位置也没用，因为此时它右侧那些位元已经是 0 了，即使再次将它们设为 0，也不会减少 $a \mid c$ 的值。

图 4.3 演示了实现这一算法的 C 语言代码。我们假定编译器会将～a &c 与 a &～c 移出循环。更为重要的是，如果支持“前导 0 计数指令”，那么就可以用下列方式初始化 m，以提升程序速度：

```
unsigned minOR(unsigned a, unsigned b,
               unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (~a & c & m) {
         temp = (a | m) & -m;
         if (temp <= b) {a = temp; break;}
      }
      else if (a & ~c & m) {
         temp = (c | m) & -m;
         if (temp <= d) {c = temp; break;}
      }
      m = m >> 1;
   }
   return a | c;
}
```

图 4.3　根据 x 与 y 的取值范围求 $x \mid y$ 的最小值

```
m = 0x80000000 >> nlz(a ^ c);
```

这种初始化方式可以跳过 a 与 c 中那些位元值均为 0 或均为 1 的位置。计算机的右移指令最好是“模 64”的，这样才能在 a^c = 0（也就是 a = c）时迅速结束循环。若是计算机不支持“前导 0 计数指令”，那么可以考虑以 a^c 为参数，使用以某种方式编写的 flp2 函数（参见 3.2 节）来初始化 m。

我们再来思考如何根据不等式（8）中给出的取值范围，求得 $x \mid y$ 的最大值。求最大值的算法与求最小值时相似，只是这次它要（从左至右）扫描的两个不等式的右边界 b、d，并寻找两数对应位元均为“1”的位置。如果找到这样的位置，那么就试着把其中一个“1”变成“0”，并将其右侧位元均设为“1”，以此提升 $b \mid d$ 的值。如果这一改动可行（也就是说，改动之后的值大于等于对应不等式的下限），那么就执行此改动，并把改动之

后的 $b \mid d$ 当成 $x \mid y$ 的上限。如果改动不可行，那么就修改 b、d 两数中刚才没有尝试过的那个数，如果两次改动均不成立，那就继续扫描下一位。此算法的 C 语言代码如图 4.4 所示，其中子表达式 `b & d` 可移至循环外，而且还可用下列方式初始化 `m` 以提升算法速度：

```
m = 0x80000000 >> nlz(b & d);
```

```
unsigned maxOR(unsigned a, unsigned b,
               unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (b & d & m) {
         temp = (b - m) | (m - 1);
         if (temp >= a) {b = temp; break;}
         temp = (d - m) | (m - 1);
         if (temp >= c) {d = temp; break;}
      }
      m = m >> 1;
   }
   return b | d;
}
```

图 4.4 根据 x 与 y 的取值范围求 $x \mid y$ 的最大值

有两种办法可以根据不等式（8）中 x、y 的取值范围求出表达式 $x \& y$ 的左右边界：代数方法和直接计算。如果用代数方法，那么根据德摩根定律可知：

$$x \& y = \neg(\neg x \mid \neg y)$$

由于我们已经知道如何求“或”操作的取值边界，而且很容易就能算出“非”操作的取值范围（$a \overset{u}{\leqslant} x \overset{u}{\leqslant} b \Leftrightarrow \neg b \overset{u}{\leqslant} \neg x \overset{u}{\leqslant} \neg a$），所以可得出下列两式：

$$\begin{aligned}&\mathrm{minAND}(a,b,c,d)\\ &= \neg\mathrm{maxOR}(\neg b,\neg a,\neg d,\neg c)\\ &\mathrm{maxAND}(a,b,c,d)\\ &= \neg\mathrm{minOR}(\neg b,\neg a,\neg d,\neg c)\end{aligned}$$

直接计算取值范围所用的代码与计算“或”操作的代码很像，如图 4.5 及图 4.6 所示。

```
unsigned minAND(unsigned a, unsigned b,
                unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (~a & ~c & m) {
         temp = (a | m) & -m;
         if (temp <= b) {a = temp; break;}
         temp = (c | m) & -m;
         if (temp <= d) {c = temp; break;}
      }
      m = m >> 1;
   }
   return a & c;
}
```

图 4.5 根据 x 与 y 的取值范围求 $x \& y$ 的最小值

```
unsigned maxAND(unsigned a, unsigned b,
                unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (b & ~d & m) {
         temp = (b & ~m) | (m - 1);
         if (temp >= a) {b = temp; break;}
      }
      else if (~b & d & m) {
         temp = (d & ~m) | (m - 1);
         if (temp >= c) {d = temp; break;}
      }
      m = m >> 1;
   }
   return b & d;
}
```

图 4.6 根据 x 与 y 的取值范围求 $x \& y$ 的最大值

通过将表达式转化为“与”、“或”、“非”等逻辑操作的组合，我们可以用代数方法求得除“异或”及“等价”外所有二元逻辑表达式的取值范围。那两个操作的取值范围之所以难算，原因就在于，如果将其表示为“与”、“或”、“非”操作的组合，那么将会出现两个同时包含 x、y 的项。例如：

$$\min_{\substack{a\leqslant x\leqslant b\\ c\leqslant y\leqslant d}}(x \oplus y) = \min_{\substack{a\leqslant x\leqslant b\\ c\leqslant y\leqslant d}}((x \& \neg y) \mid (\neg x \& y))$$

我们本来想寻找一个 x 与一个 y，令整个“或”表达式的结果最小，然而在未经证明的情况下，我们不敢肯定（实际上事后证

明是可以的）找到的 x 和 y 值是否可以分别使“或”操作的两个操作数取至其最小值。㊀

下列表达式可以计算“异或”操作的取值范围：

$$\mathrm{minXOR}(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c},\boldsymbol{d})=\mathrm{minAND}(\boldsymbol{a},\boldsymbol{b},\neg\boldsymbol{d},\neg\boldsymbol{c})\mid\mathrm{minAND}(\neg\boldsymbol{b},\neg\boldsymbol{a},\boldsymbol{c},\boldsymbol{d})$$

$$\begin{aligned}\mathrm{maxXOR}(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c},\boldsymbol{d})=\mathrm{maxOR}(&\mathbf{0},\mathrm{maxAND}(\boldsymbol{a},\boldsymbol{b},\neg\boldsymbol{d},\neg\boldsymbol{c}),\\&\mathbf{0},\mathrm{maxAND}(\neg\boldsymbol{b},\neg\boldsymbol{a},\boldsymbol{c},\boldsymbol{d}))\end{aligned}$$

用直接计算的方式求 minXOR 及 maxXOR 非常简单。计算 minXOR 的代码与图 4.3 中计算 minOR 的代码一样，只要去掉两个 break 语句并把返回值变成 a^c 就行了。而计算 maxXOR 的代码也与图 4.4 中计算 maxOR 的代码类似，只需用如下程序替换 if 语句体中的 4 行代码并把返回值改为 b^d 即可。

```
temp = (b - m) | (m - 1);
if (temp >= a) b = temp;
else {
   temp = (d - m) | (m - 1);
   if (temp >= c) d = temp;
}
```

带符号数逻辑操作的取值范围

两个带符号数经过逻辑表达式求值之后的取值范围与无符号数比起来，又复杂了很多。如果 0 包含在 a 至 b 或 c 至 d 之间，那么计算过程就是无规律的。表 4.1 列出了一种计算表达式 $x \mid y$ 上下限的方式。其中“+”表示该列顶部的那个边界值大于等于 0，而“−”则表示该值小于 0。标有“minOR（带符号）”的这一列，其中的各个表达式用来计算 $x \mid y$ 的下限，而标有“maxOR（带符号）”的这一列，其中的各个表达式则用来计算 $x \mid y$的上限。用程序来实现这个算法时，可以设置一个变量，用变量值 0 至 15 分别表示 $\boldsymbol{a}$、$\boldsymbol{b}$、$\boldsymbol{c}$、$\boldsymbol{d}$ 的 16 种取值组合，然后让“switch”语句根据该变量的值来分别执行每种取值组合下的具体算法。请注意，0 至 15 这 16 个值不会全部用到，因为不可能出现像 $\boldsymbol{a}>\boldsymbol{b}$ 或 $\boldsymbol{c}>\boldsymbol{d}$ 这种取值组合。

下列关系

$$\boldsymbol{a}\leqslant\boldsymbol{x}\leqslant\boldsymbol{b}\Leftrightarrow\neg\boldsymbol{b}\leqslant\neg\boldsymbol{x}\leqslant\neg\boldsymbol{a}$$

对于带符号数依然成立，所以可使用代数方法推广表 4.1 中的结果，以求出（除“异或”及“等价”外）其他逻辑表达式的取值范围。此处将这些问题留给读者完成。

表 4.1　由无符号的 minOR 及 maxOR 函数求带符号版本的函数值

a	b	c	d	**minOR**（带符号数）	**maxOR**（带符号数）
−	−	−	−	minOR（a，b，c，d）	maxOR（a，b，c，d）
−	−	−	+	a	−1
−	−	+	+	minOR（a，b，c，d）	maxOR（a，b，c，d）

㊀ 因为两个操作数 $x\ \&\neg y$ 与 $\neg x\ \&y$ 中都有 x、y，所以这两项存在联动关系。在未经证明的情况下，我们担心如果让“或”操作左方的表达式取至最小值，会不会影响到其右方表达式的取值，反之亦然。——译者注

（续）

a	*b*	*c*	*d*	minOR（带符号数）	maxOR（带符号数）
−	+	−	−	*c*	−1
−	+	−	+	min（*a*，*c*）	maxOR（0，*b*，0，*d*）
−	+	+	+	minOR（*a*，0xFFFFFFFF，*c*，*d*）	maxOR（0，*b*，*c*，*d*）
+	+	−	−	minOR（*a*，*b*，*c*，*d*）	maxOR（*a*，*b*，*c*，*d*）
+	+	−	+	minOR（*a*，*b*，*c*，0xFFFFFFFF）	maxOR（*a*，*b*，0，*d*）
+	+	+	+	minOR（*a*，*b*，*c*，*d*）	maxOR（*a*，*b*，*c*，*d*）

4.4　习题

1. 已知两个无符号整数 x、y 的取值范围如下，请问二者之差 $x-y$ 的取值范围是什么？

$$0 \overset{u}{\leqslant} x \overset{u}{\leqslant} b$$

$$0 \overset{u}{\leqslant} y \overset{u}{\leqslant} d$$

2. 如果某台电脑支持“前导 0 计数指令”，那么在 a = 0 或 c = 0 的情况下，应该如何简化图 4.4 中的 maxOR 函数呢？

第 5 章

位计数

5.1 统计值为“1”的位元数

IBM 的 Stretch 计算机[㊀](大约问世于 1960 年)有办法算出一个字组中值为“1”的位元个数，同时也能求出前导 0 的个数，这是因为，CPU 在执行每一种逻辑操作时，都会附带着算出上述两个值。前者有时也叫“种群计数”(population count)函数(例如 Stretch 及 SPARCv9 架构的 CPU 就有能够实现该功能的指令)。

如果计算机不支持此指令，那么可以用另外一种好办法来计算值为“1”的位元个数：首先，把邻近的两个位元值编组为 1 个位段，将这两个位元的值相加，把结果放到这个宽度为 2 的位段中。然后把邻近的两个二位元位段相加，将结果放在宽度为 4 的位段中，以此类推。[RND] 一书更加详细地讨论了此技巧。图 5.1 演示了计算过程，我们要统计的是第 1 行那个计算机字组中值为“1”的位元个数，最终计算结果列在了最末一行(计算结果用十进制表示就是 23)。

此算法应用了“分而治之”策略[㊁]，将原始问题(统计 32 个位元中值为“1”的位元个数)化为两个小问题(统计 16 个位元中值为“1”的位元个数)，分别将其解决，然后再将结果合并(本例中使用加法运算来合并)。递归运用该策略，将宽度为 16 的位段拆解为两个宽度为 8 的位段。

在本例中，分解到不能再分的若干小问题(也就是统计相邻两个位元中值为“1”的位元数)可以并行求解，然后反复运用并行计算的方式，把相邻两个位段的统计结果合并起来，这样经过一定的步骤，即可求得最终结果。在这个例子中，算法需要 $\log_2(32)$(也就是 5 个步骤)就可求出最终结果了。

㊀ 又叫 IBM 7030，是 IBM 首台晶体管巨型计算机。详情参见：http://en.wikipedia.org/wiki/IBM_7030_Stretch。——译者注

㊁ Divide and Conquer，简称 D&C，中文称“分治法”，是一种算法范式，它把一个复杂问题递归分成两个或更多相同或相似的子问题，直到最后的子问题简单到能够直接求解为止，然后将子问题的解合并起来，就得出了原问题的解。详情参见：http://zh.wikipedia.org/wiki/分治法。——译者注

还有一些广为人知的分治法，例如“二分搜索法”[一]、“快速排序法”[二]等。7.1 节会讲到如何反向排列字组中的位元，该过程也会用到分治法。

图 5.1 中的算法用 C 语言写出来就是：

```
x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF);
```

第 1 行本来写成（x & 0xAAAA AAAA)≫1 比较自然，但是笔者却用了（x≫1）& 0x5555 5555 这种写法，这么做可以避免在寄存器中生成两种大常数。因为如果计算机没有“and not”指令，则前面那种做法会多耗费一条指令。[三]其他各行的写法也基于这一考量。.

最后一行的“与”运算明显多余，而且在位段求和操作肯定不会导致向邻近位段进位时，其余各行中的“与”运算也可省略。此外，第 1 行代码还有一种写法，可以少用 1 条指令。综上所述，可以得出图 5.2 中的这段代码，它只需 21 条指令，而且没有分支。

1 0 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1
01 10 10 00 01 01 00 10 01 10 10 01 10 10 10 10
0011 0010 0010 0010 0011 0011 0100 0100
00000101 00000100 00000110 00000100
0000000000001001 0000000000001110
00000000000000000000000000010111

图 5.1　用分治法统计字组中值为“1”的位元数

[一] binary search，也称为二分查找法，是一种在有序数组中查找特定元素的搜索算法。搜索过程从数组中间的元素开始，如果中间元素正好是要查找的元素，则结束搜索；如果待查元素大于或小于中间元素，则在数组大于或小于中间元素的那一半中查找，并继续跟那一半的中间元素比较，递归执行此过程，直至找到。如果在某一步中折半后的范围为 0，则代表找不到。该算法每次都将搜索范围缩小为原来的一半。详情参见：http：//zh. wikipedia. org/wiki/折半搜索算法。——译者注

[二] quicksort，由英国计算机学者托尼·霍尔（Sir Charles Antony Richard Hoare，缩写为 C. A. R. Hoare，昵称 Tony Hoare，生于 1934 年）研发的一种排序算法，排序 n 个项目平均要 O（nlogn）次比较。其步骤为：从数组中挑出一个“基准”（pivot）元素，重新排序数组，把所有比基准值小的元素都放在基准值前，把所有比基准值大的元素都放在基准值后（与基准值相同的数可随意放到任意一边）。然后在基准值左右两个分区内继续分别寻找新的基准值，递归执行此过程，直至每个小分区的长度均小于 2 为止。详情参见：http：//zh. wikipedia. org/快速排序。——译者注

[三] 依本书 1.2 节所说，由于大部分带立即数的 RISC 指令都只支持 16 位常量，所以若某常数比较“大”，也就是用 16 个位元无法容纳，则通常要先花两条指令将其载入寄存器，然后再参与其他运算。本例中，若写成“(x & 0x5555 5555) + (x & 0xAAAA AAAA) ≫1；”，则计算到“x & 0xAAAA AAAA”时，必须把先前装入寄存器中的“大常数”0x5555 5555 取反，变成 0xAAAA AAAA，然后再和 x 取与。若有“and not”指令，此操作可一步完成，否则必须先取反，再取与，这样就比“(x≫1) & 0x5555 5555”多花 1 条指令了。有鉴于此，作者设法减少代码用到的常量种数，尽量使用重复的常数。——译者注

```
int pop(unsigned x) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
   x = (x + (x >> 4)) & 0x0F0F0F0F;
   x = x + (x >> 8);
   x = x + (x >> 16);
   return x & 0x0000003F;
}
```

图 5.2 统计字组中值为"1"的位元数

第 1 行中赋给 x 的值是根据下列公式的前两项而来。这个公式看上去相当奇特：

$$\mathrm{pop}(x)=x-\left\lfloor\frac{x}{2}\right\rfloor-\left\lfloor\frac{x}{4}\right\rfloor-\cdots-\left\lfloor\frac{x}{2^{31}}\right\rfloor \tag{1}$$

等式（1）中的 x 必须大于等于 0。如果把 x 视为无符号数，那么用 31 条移位长度为 1 的"立即数右移"指令，再搭配 31 条减法指令，即可求出等式（1）的值。图 5.2 中的这段代码利用此公式的前两项来并行计算每一组宽度为 2 的位段。

我们以 4 位字组为例，用一种简单的办法证明等式（1）。设字组中的 4 个位元分别为 $b_3b_2b_1b_0$，其中 $b_i=0$ 或 1。那么：

$$\begin{aligned}
x-\left\lfloor\frac{x}{2}\right\rfloor-\left\lfloor\frac{x}{4}\right\rfloor-\left\lfloor\frac{x}{8}\right\rfloor &= b_3\cdot 2^3+b_2\cdot 2^2+b_1\cdot 2^1+b_0\cdot 2^0\\
&\quad-(b_3\cdot 2^2+b_2\cdot 2^1+b_1\cdot 2^0)\\
&\quad-(b_3\cdot 2^1+b_2\cdot 2^0)\\
&\quad-(b_3\cdot 2^0)\\
&=b_3(2^3-2^2-2^1-2^0)+b_2(2^2-2^1-2^0)+b_1(2^1-2^0)+b_0(2^0)\\
&=b_3+b_2+b_1+b_0
\end{aligned}$$

此外还有一种证明等式（1）的方法。如果用二进制表示非负无符号整数 x，那么其第 i 位的值就是：

$$b_i=\left\lfloor\frac{x}{2^i}\right\rfloor-2\left\lfloor\frac{x}{2^{i+1}}\right\rfloor$$

把 0 至 31 之间的 32 个整数分别作为 i 的值代入上式，然后把求出的 b_0 至 b_{31} 这 32 个数加起来，即可证明等式（1）。由于 $x<2^{32}$，所以当 $i=31$ 时，上式中最后一项的值是 0。

等式（1）也可以推广至其他进制的数。例如在十进制[㊀]的情况下：

$$\mathrm{sum\_digits}(x)=x-9\left\lfloor\frac{x}{10}\right\rfloor-9\left\lfloor\frac{x}{100}\right\rfloor-\cdots$$

上式中的项可一直往下续写，直到 0 为止。此式的证明方法与二进制版本实质相同。

图 5.2 还有一种写法，那就是将等式（1）套用在四进制的情况下，然后据此把第 2 行可执行代码改写为：

㊀ 原文为 base ten，即"以 10 为基"或"以 10 为底"，这里 base 后面的数字就是每一"位"上可以出现的数字个数。例如十进制数的每一位可以出现 0～9 这 10 个数字，而且每一位的权重都是 10 的整数幂，所以叫"以 10 为基"。详情参见：http：//zh. wikipedia. org/wiki/进位制。——译者注

```
x = x - 3*((x >> 2) & 0x33333333)
```

改写后的代码与原代码所需指令数相同（都需要6条指令），而且还要求计算机能够快速执行“乘以3”的指令。

HAKMEM备忘录［HAK，item 169］中还有一种算法，能够算出字组中值为“1”的位元数。它先根据等式（1）的前3项把字组切割成长度为3的位段，并将每个位段中值为“1”的位元个数写入该位段。然后将相邻两个宽度是3的位段合并为一个宽度是6的位段，最后把字组的值与63取模，以此来合并那些宽度为6的位段。该算法的C语言代码如下（其中以0开头的长串整数均为8进制写法）：

```
int pop(unsigned x) {
   unsigned n;

   n = (x >> 1) & 033333333333;       // Count bits in
   x = x - n;                         // each 3-bit
   n = (n >> 1) & 033333333333;       // field.
   x = x - n;
   x = (x + (x >> 3)) & 030707070707; // 6-bit sums.
   return x%63;                       // Add 6-bit sums.
}
```

最后一行用到了“无符号模除”函数。（如果字组长度是3的倍数，那么带不带符号无所谓。）模除函数为什么能够对x中长度为6的若干个位段求和呢？其实只要把x写成64进制数，就立刻能明白其中的原因了。不论是把一个b进制整数（$b \geqslant 3$）直接除以$b-1$，还是把该数各个位置上的数字加起来再除以$b-1$，其得到的余数都一样，而且这个余数显然比$b-1$小。由于本例中，x各个位元上的值加起来肯定小于等于32，所以$\mathrm{mod}(x,63)$算出来的值也肯定比x中各个位元值之和要小，换句话说，该值也就是一开始的那个x中值为“1”的位元个数。

在DEC PDP-10中，该算法仅需10条指令，因为此计算机可以把内存中的全字直接用作模除指令的操作数。在拥有基本RISC指令集的计算机中，假定CPU能用1条指令实现无符号模除操作（同时假定该指令不能直接引用全字常数或内存中的操作数），那么该算法需要13条指令。这种算法可能不会很快，因为除法操作基本上都执行得比较慢。此外，仅通过扩展算法中的常数，是无法将其直接套用在64位字组上的，该算法最多能扩展至位宽为62的字组。

也可以用下列代码替换return语句，这样的话，该算法在大部分电脑中将会运行得更快。不过这种写法看上去不太优雅（以0开头的数依然是8进制写法）：

```
return ((x * 0404040404) >> 26) +  // Add 6-bit sums.
       (x >> 30);
```

HAKMEM算法也存在一种变体：用等式（1）统计出每个长度为4的位段中值为“1”的位元个数，这样可以并行计算好8个长度为4的位段［Hay1］。然后直接把相邻两个长度为4的位段合并为1个长度为8的位段。最后通过乘以0x0101 0101的方式对这4个字节求和。上述算法的代码如下：

```
int pop(unsigned x) {
   unsigned n;

   n = (x >> 1) & 0x77777777;        // Count bits in
   x = x - n;                        // each 4-bit
   n = (n >> 1) & 0x77777777;        // field.
   x = x - n;
   n = (n >> 1) & 0x77777777;
   x = x - n;
   x = (x + (x >> 4)) & 0x0F0F0F0F;  // Get byte sums
   x = x*0x01010101;                 // Add the bytes
   return x >> 24;
}
```

在具备基本RISC指令集的计算机中，该算法需要19条指令。其中前6行用1条“寄存器移动”指令（move register instruction）即可实现，而且还可以把掩码0x7777 7777载入寄存器中，这样那种接受两个寄存器作参数的指令（register-to-register instruction）就可以多次引用它了。此外，右移指令大多只是移动1个位置。

图5.3演示了另外一种位计数法［Weg，RND］：反复将最右侧的“1”变为“0”，直到整个数字的值变成0为止。如果值为“1”的位元个数不多，那么该算法执行起来很快，只需 $2+5\text{pop}(x)$ 条指令即可。

```
int pop(unsigned x) {
   int n;

   n = 0;
   while (x != 0) {
      n = n + 1;
      x = x & (x - 1);
   }
   return n;
}
```

图 5.3

还有一个与之对称的算法，适用于值为“1”的位元个数较多的情况。这个算法以x = x | (x + 1) 的方式反复将最右侧的“0”变为“1”，直至整个数字的所有位元都变成“1”为止(此时这个数的值是－1)。然后，返回 $32-n$。(如果想直接返回 n 而不想计算 $32-n$ 的话，那么可以先把 x 取反再套用算法，或是将 n 的初始值设为32，然后倒着向下计数。)

有一种算法看上去很神奇：将 $\boldsymbol{x}$ 循环左移1位，执行31次这样的操作，然后把32项加起来［MM］，这样求出的总和居然是 $\text{pop}(\boldsymbol{x})$ 的相反数！也就是说：

$$\text{pop}(\boldsymbol{x}) = -\sum_{i=0}^{31}(\boldsymbol{x} \overset{rot}{\ll} i) \tag{2}$$

加法的结果要按照字组长度裁剪，而且最终值应视为补码。虽然这个算法的思路很新颖，然而在大多数计算机上都不实用。因为要循环31次，所以整个算法需要63条指令才行，此外还得考虑循环控制指令的开销。

为了让大家理解等式（2）的工作原理，我们来思考一下：如果字组 $\boldsymbol{x}$ 中只有1个位元的值是“1”，那么按照上述算法，这个位元于循环左移的过程中，在全部32个位置上都会各出现一次，将这32个数字相加之后，结果的每一个位元必然都是“1”，而这样值为－1的字组恰好是 $\text{pop}(x)$ 的相反数。现在以长度为6的字组来演示这一过程，假设 $\boldsymbol{x}=\mathbf{001001}$（二进制）：

$$\begin{array}{ll} \mathbf{001001} & \boldsymbol{x} \\ \mathbf{010010} & \boldsymbol{x} \overset{rot}{\ll} 1 \end{array}$$

$$
\begin{array}{ll}
100100 & x \overset{rot}{\ll} 2 \\
001001 & x \overset{rot}{\ll} 3 \\
010010 & x \overset{rot}{\ll} 4 \\
100100 & x \overset{rot}{\ll} 5
\end{array}
$$

用循环右移当然也能算出正确结果。

只要把等式（1）改写一下，就能够明显看出：它的算法很像上面这种“循环移位并相加”算法：

$$\text{pop}(x) = x - \sum_{i=1}^{31} (x \overset{u}{\gg} i)$$

这样会比等式（2）的算法稍微好些：因为它用的是比循环移位指令更常见的右移指令，而且一旦发现待右移的量已经是 0 了，就可以提前终止循环。这种算法减少了循环控制代码，而且有可能节省几轮循环时间。图 5.4 对比了这两种算法。

```
int pop(unsigned x) {
   int i, sum;

// Rotate and sum method          // Shift right & subtract

   sum = x;                       // sum = x;
   for (i = 1; i <= 31; i++) {    // while (x != 0) {
      x = rotatel(x, 1);          //    x = x >> 1;
      sum = sum + x;              //    sum = sum - x;
   }                              // }
   return -sum;                   // return sum;
}
```

图 5.4 两个相似的位计数算法

还有一种算法不及它们有趣，不过其效率比得上本节中介绍的其他 pop(x) 算法。该算法预先算好 0 到 255 之间的 pop(x) 值并将其存入表格，然后根据待计算的数值来查表，并把 4 次查表的结果加起来。可以用无分支代码来实现此算法：

```
int pop(unsigned x) {                 // Table lookup.
   static char table[256] = {
      0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
      ...
      4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8};
   return table[x         & 0xFF] +
          table[(x >>  8) & 0xFF] +
          table[(x >> 16) & 0xFF] +
          table[(x >> 24)];
}
```

[HAK] 中的 Item167 介绍了一种短小的算法，可以单独统计出寄存器里靠右对齐的 9 位数中值为“1”的位元个数。此算法仅适用于寄存器位宽大于等于 36 的电脑。下面列出的这个版本可以用在 32 位机上，只不过仅适用于 8 位元的数。

```
x = x * 0x08040201;  // Make 4 copies.
x = x >> 3;          // So next step hits proper bits.
x = x & 0x11111111;  // Every 4th bit.
x = x * 0x11111111;  // Sum the digits (each 0 or 1).
x = x >> 28;         // Position the result.
```

适用于7位量的写法是：

```
x = x * 0x02040810;  // Make 4 copies, left-adjusted.
x = x & 0x11111111;  // Every 4th bit.
x = x * 0x11111111;  // Sum the digits (each 0 or 1).
x = x >> 28;         // Position the result.
```

在上述两种算法中，最后两步可以改写为：将x与15取模，并把余数赋给x。

这两种算法都不是很理想，大部分程序员还是会选用查表法。不过，上述两种算法中的后者，可以推广至64位算术，这样的话，它就可以在能够快速执行乘法的64位计算机上使用了。扩展后的版本可以计算15位的量。（笔者不觉得存在一种适用于16位二进制数的相似算法，除非计算之前可以肯定那16个位元不全是"1"。）很多C语言编译器，不论新旧，都会提供一种存放64位整数的扩充数据类型，名叫long long，它现在已经包含于官方C99标准内了。后缀为ULL的值，表示unsigned long long型常数。

```
int pop(unsigned x) {
   unsigned long long y;
   y = x * 0x0002000400080010ULL;
   y = y & 0x1111111111111111ULL;
   y = y * 0x1111111111111111ULL;
   y = y >> 60;
   return y;
}
```

5.1.1 两个字组种群计数的和与差

要计算 $\text{pop}(\boldsymbol{x})+\text{pop}(\boldsymbol{y})$（假设计算机中没有"种群计数指令"），可以先按图5.2中的前两行代码分别计算 $\boldsymbol{x}$、$\boldsymbol{y}$，然后把 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 相加，将相加之后的结果代入算法最后3步。执行完图5.2的前两行之后，$\boldsymbol{x}$ 与 $\boldsymbol{y}$ 均包含8个长度是4的位段，而每个位段的最大值是4，所以把两个位段加起来之后，最大结果才是8，不可能向邻近字段溢出。（实际上，3个这样的字段加起来都不会溢出。）

该思路也可用于计算两个字组种群计数之差。要计算 $\text{pop}(\boldsymbol{x})-\text{pop}(\boldsymbol{y})$，可使用下式：

$$\begin{aligned}\text{pop}(\boldsymbol{x})-\text{pop}(\boldsymbol{y})&=\text{pop}(\boldsymbol{x})-(32-\text{pop}(\bar{\boldsymbol{y}}))\\&=\text{pop}(\boldsymbol{x})+\text{pop}(\bar{\boldsymbol{y}})-32\end{aligned}$$

然后，根据前面讲过的技巧算出 $\text{pop}(\boldsymbol{x})+\text{pop}(\bar{\boldsymbol{y}})$。图5.5列出了这段代码，它共需32条指令，而如果把 $\boldsymbol{x}$、$\boldsymbol{y}$ 分别代入图5.2，然后再相减，则需要43条指令。

5.1.2 比较两个字组的种群计数

有时我们想在不计算具体数值的情况下比较两个字组的种群计数，看谁更大一些。那么有没有办法不经计算就可以比较出二者种群计数的大小呢？按照图5.5，算出两个字

组种群计数之差，然后和0比较，这算是一种办法。不过我们想要的是另外一种办法，因为有时候可以预先得知在待比较的数中，值为“1”的位元个数并不多，或是两个字组中值为“1”的那些位元存在某种密切关联。

```
int popDiff(unsigned x, unsigned y) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
   y = ~y;
   y = y - ((y >> 1) & 0x55555555);
   y = (y & 0x33333333) + ((y >> 2) & 0x33333333);
   x = x + y;
   x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
   x = x + (x >> 8);
   x = x + (x >> 16);
   return (x & 0x0000007F) - 32;
}
```

图 5.5 计算 $\text{pop}(\boldsymbol{x})-\text{pop}(\boldsymbol{y})$

新算法的思路是：反复将两个数中的1个位元清零，直到其中一个数的所有位元都变成“0”为止，另外一个数就是种群计数较大的那个。执行上述过程前，先把 $\boldsymbol{x}$、$\boldsymbol{y}$ 中对应位置都为“1”的位元清零，以便提高算法在平均状况及最差状况下的执行速度。图5.6列出了这个算法的代码，如果 $\text{pop}(\boldsymbol{x})<\text{pop}(\boldsymbol{y})$，则会返回负值；如果 $\text{pop}(\boldsymbol{x})=\text{pop}(\boldsymbol{y})$，则会返回0；在 $\text{pop}(\boldsymbol{x})>\text{pop}(\boldsymbol{y})$ 的情况下，返回值为正数（也就是1）。

```
int popCmpr(unsigned xp, unsigned yp) {
   unsigned x, y;
   x = xp & ~yp;                  // Clear bits where
   y = yp & ~xp;                  // both are 1.
   while (1) {
      if (x == 0) return y | -y;
      if (y == 0) return 1;
      x = x & (x - 1);            // Clear one bit
      y = y & (y - 1);            // from each.
   }
}
```

图 5.6 比较 $\text{pop}(\boldsymbol{x})$ 与 $\text{pop}(\boldsymbol{y})$ 的大小

将 $\boldsymbol{x}$、$\boldsymbol{y}$ 中对应位置均为1的位元清零后，两数值为“1”的位元数加起来至多只会有32个。因此，两数中种群计数较小的那一个最多只会有16个值为“1”的位元。因此，图5.6中的循环最多也就只会执行16次而已。这样算下来，在最坏的情况下，共需119（$16\times7+7$）条基本RISC指令。有人曾经用均匀分布的32位随机整数模拟测试了该算法，结果表明：除去对应位置均为“1”的那些位元之后，两数之中种群计数较小的那个数平均含有6.186个值为“1”的位元。这也就是说，对于随机分布的32位输入值，该算法平均要花费50条指令才能执行完，它的速度比不上图5.5中的那种算法。只有在去掉了对应位置均为“1”的那些位元后，$\boldsymbol{x}$ 或 $\boldsymbol{y}$ 中值为“1”的位元数小于等于3时，此算法才能胜过图5.5中的那个。

5.1.3 统计数组中值为“1”的位元数

在没有种群计数指令的前提下，要想统计全字数组（array 或 vector）中值为“1”的位元数，最简单的方法就是用图 5.2 中的算法，把每个字组的种群计数都统计出来，最后把结果相加。我们把这种算法称为“笨办法”[㊀]。如果不算循环控制指令，也不考虑生成常数并将其加载到数组中所用的指令，那么处理每个字组要花费 16 条指令：其中 15 条用于实现图 5.2 中的代码，还有 1 条用于和其他字组的统计结果相加。此处假定算法是以内联（in line）形式而非函数调用形式嵌入程序中的，并且所有掩码都是在循环之外加载的，还需假设计算机的寄存器数量足够多，可以保存下计算中用到的那些数值。

另外一种方法是，以 3 个字组为单位，将其代入图 5.2 中的前两行可执行代码中，然后把 3 个局部统计结果加总。由于每个 4 位元位段的最大值是 4，所以 3 个分散在单个字组中的位段加起来，最大值也不过 12，不会向邻近位段溢出。加总长度为 8 和 16 的位段时也是这样。将该算法编码并编译后可以看出，与刚才那种朴素算法相比，它所需的基本 RISC 指令数大约减少了 20%，而这一优势却让算法中多出来的那些辅助指令（housekeeping instruction）给抵消了。此处我们不再继续深究此算法了，因为还有一种算法比它好得多。

这个好办法似乎是 Robert Harley 与 David Seal 于 1996 年左右发明的 [Seal1]。此算法基于一种名为“进位保留加法器”（carry-save adder，CSA）的电路，它也叫 3∶2 压缩器[㊁]。CSA 是由一系列相互独立的“全加器”[㊂]组成的 [H & P]，通常用于实现二进制乘法器的电路。

全加器的逻辑用布尔代数表示出来就是：

$$h \leftarrow ab+ac+bc=ab+(a+b)c=ab+(a \oplus b)c,$$
$$l \leftarrow (a \oplus b) \oplus c$$

其中 a、b、c 均为 1 位输入值，l 是低位输出（也就是求和结果），h 是高位输出（也就是加法产生的进位）。将第 1 行的 $a+b$ 换成 $a \oplus b$ 是正确的，因为如果 a 和 b 都是 1，那么 ab 这一项的值也是 1，而这正是整个表达式的值。如果把 $a \oplus b$ 的结果临时保存起来，那么用 5 条指令即可实现全加器逻辑，而且（在 32 位机上）能同时计算 32 个位元。我们将这 5 条指令记为 CSA(h,l,a,b,c)。这是一个“宏”（macro），其中 h 与 l 表示输出值。

CSA 操作的一种用法是，将数组 A 中的元素每 3 个编为 1 组，然后将每组中的 3 个

㊀ naive method，指没有经过加工、优化、完善等步骤的朴素算法。——译者注

㊁ 进位保留加法器与 3∶2 压缩器的详细介绍请参见：http://en.wikipedia.org/wiki/Carry-save_adder 与 http://en.wikipedia.org/wiki/Adder_(electronics)#3∶2_compressors。——译者注

㊂ full adder，是一种能够接受两个二进制数及一个低位进位的加法器。详情参见：http://zh.wikipedia.org/wiki/加法器#.E5.85.A8.E5.8A.A0.E5.99.A8。——译者注

字归并为两个，然后再统计这两个字的种群计数。在循环过程中，将两个字组的种群计数相加。执行完循环后，数组中总的种群计数就等于 CSA 高位累加的种群计数值乘以 2，再加上低位累加的种群计数。

设 n_c 为 CSA 操作所需的指令数，而 n_p 为计算 1 个字组的种群计数所需的指令数。在常见的 RISC 架构计算机中，$n_c=5$，$n_p=15$。忽略从数组中载入元素及循环控制所用的指令（此类任务所需的代码在各个计算机上或许颇为不同），那么用上面讨论的算法，统计 1 个字组平均需要 $(n_c+2n_p+2)/3\approx 12.33$ 条指令（"+2"是为了算上循环内使用的两个加法指令），而与之相比，未优化的算法则需 16 条指令。

以另外一种方式使用 CSA 操作，可以写出效率更好而且稍微精简一些的代码。图 5.7 演示了新算法，它统计每个字组平均需要 $(n_c+n_p+1)/2=10.5$ 条指令（不计循环控制与加载指令）。在这段代码中，CSA 操作会展开成：

```
u = ones ^ A[i];
v = A[i+1];
twos = (ones & A[i]) | (u & v);
ones = u ^ v;
```

我们要依赖编译器将那些重复的载入操作合并，这样此算法的代码效率才会提升。

```
#define CSA(h,l, a,b,c) \
   {unsigned u = a ^ b; unsigned v = c; \
      h = (a & b) | (u & v); l = u ^ v;}

int popArray(unsigned A[], int n) {

   int tot, i;
   unsigned ones, twos;

   tot = 0;                      // Initialize.
   ones = 0;
   for (i = 0; i <= n - 2; i = i + 2) {
      CSA(twos, ones, ones, A[i], A[i+1])
      tot = tot + pop(twos);
   }
   tot = 2*tot + pop(ones);

   if (n & 1)                    // If there's a last one,
      tot = tot + pop(A[i]);     // add it in.

   return tot;
}
```

图 5.7 求数组的种群计数，每次处理两个字组元素

CSA 操作还有一些用法，可以继续降低统计数组种群计数所需的指令个数，把它们用电路图画出来就更容易理解了。比如，图 5.8 所示就是将一个算法的循环代码画成了电路图。该循环每次能够处理 8 个数组元素，并将其种群计数分别压缩至标有 *eights*、*fours*、*twos*、*ones* 的四个量中。下一轮循环将会把 *fours*、*twos*、*ones* 都重新填入 CSA 指令，并统计字组 *eights* 的种群计数，将其中值为"1"的位元数累加起来。处理完全部数组元素之后，总的种群计数就是：

$$8\text{pop}(eights)+4\text{pop}(fours)+2\text{pop}(twos)+\text{pop}(ones)$$

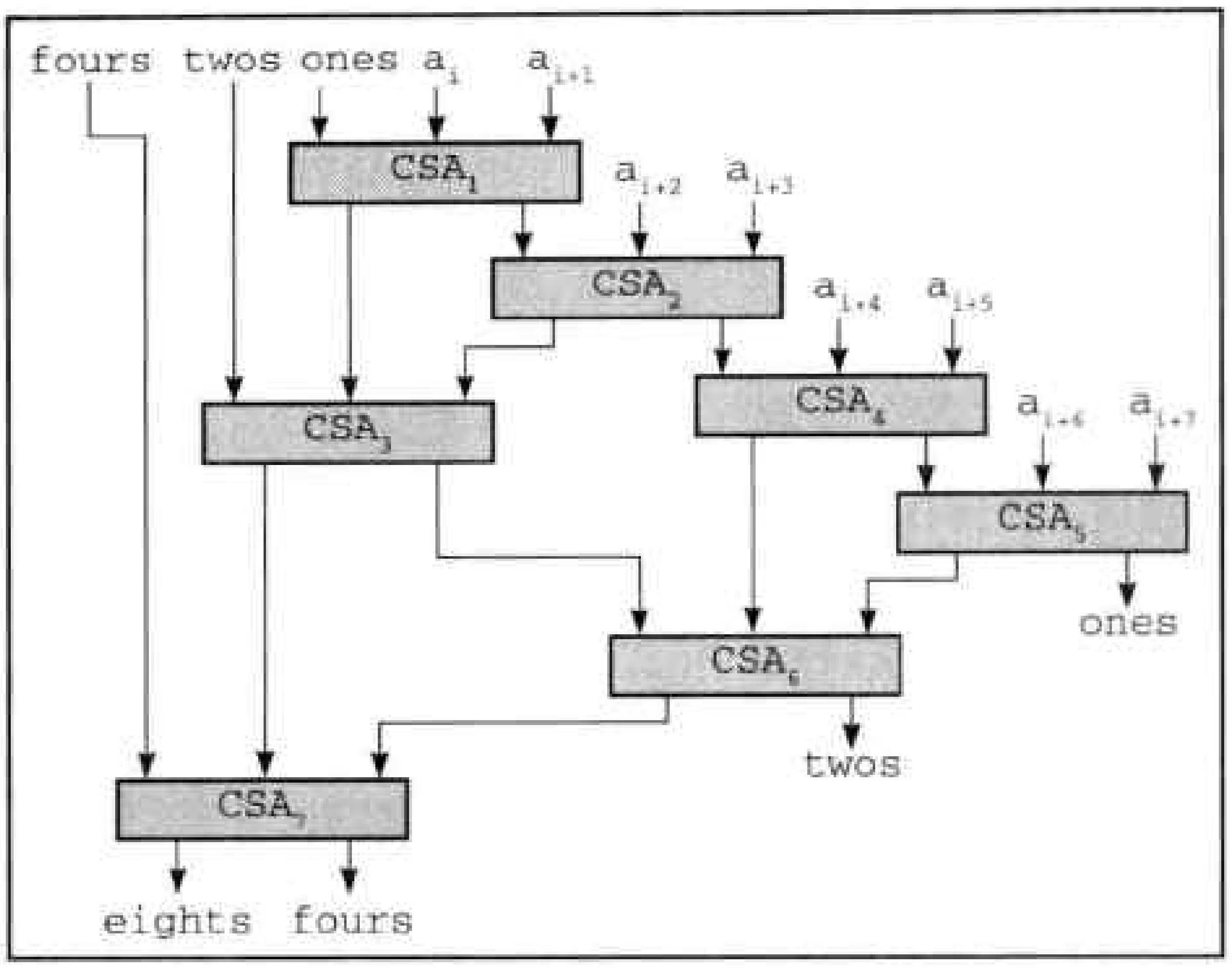


图 5.8 用电路图表示统计数组种群计数的算法

图 5.9 列出了本算法的代码，其中使用了图 5.7 所定义的 CSA 宏。图 5.8 中，CSA 右下方出现的数字就表示图 5.9 调用 CSA 宏的次序。除去加载数组元素及控制循环的开销，这段代码的循环体每处理数组中的一个字组，需要花 $(7n_c+n_p+1)/8=6.375$ 条指令。

```
int popArray(unsigned A[], int n) {

   int tot, i;
   unsigned ones, twos, twosA, twosB,
      fours, foursA, foursB, eights;

   tot = 0;                     // Initialize.
   fours = twos = ones = 0;

   for (i = 0; i <= n - 8; i = i + 8) {
      CSA(twosA, ones, ones, A[i], A[i+1])
      CSA(twosB, ones, ones, A[i+2], A[i+3])
      CSA(foursA, twos, twos, twosA, twosB)
      CSA(twosA, ones, ones, A[i+4], A[i+5])
      CSA(twosB, ones, ones, A[i+6], A[i+7])
      CSA(foursB, twos, twos, twosA, twosB)
      CSA(eights, fours, fours, foursA, foursB)
      tot = tot + pop(eights);
   }
   tot = 8*tot + 4*pop(fours) + 2*pop(twos) + pop(ones);

   for (i = i; i < n; i++)      // Simply add in the last
      tot = tot + pop(A[i]);    // 0 to 7 elements.
   return tot;
}
```

图 5.9 与算法对应的代码

CSA 指令的连接次序也可以和图 5.8 有所不同。比如，为了提升算法的并行处理能力，我们可以把头 3 个数组元素填入一个 CSA 中，然后把接下来的 3 个数组元素填入另

外一个 CSA 中，这样就可以同时计算两个 CSA 了。或者，我们重新排列 CSA 宏里 3 个操作数的次序，这么做也能提升算法的并行处理能力。根据图 5.8 所示方案，大家很容易就能构建出一段程序，仅用前 3 个 CSA 来处理数组中的 4 个元素，然后，可将此思路扩展，以编写出一次能处理 16 个或更多数组元素的代码。这套方案还摊薄了数据加载的压力：有些计算机同一时间所能加载的数据量比较少，在此类计算机上，该方案就能体现出优势了。

可以将图 5.8 所示方案推广，以便进一步减少种群计数指令的调用次数。为了说明这个新思路，我们需要一个大小为 $m\times2$ 的数组。这个数组用来存放每一对被打上 *ones*、*twos*、*fours* 等标记的变量。如果数组长度是 n，那么只要选择大于等于$\lceil \log_2(n+1)\rceil+1$ 的 m 值就够了（若 m 等于 31，则适用于 32 位字节地址空间所能够容下的任意长度数组）。此外还需要一个长度为 m 的字节数组，用于记录 $m\times2$ 数组中的每一行已经存放了多少个值（是 0 个、1 个还是 2 个）。程序将每两个元素编为一组，然后把它们与一个标注为 *ones* 的变量填入 CSA 中。为方便计算，一般可以把这个 *ones* 变量放在 $m\times2$ 数组中的 [0, 0] 位置上。在内层循环中，向下扫描 $m\times2$ 数组，直至找到一个元素占用量小于 2 的行（通常用不了多久就能找到），然后把刚才 CSA 所输出的 *twos* 值放入该行。如果用来存放 *twos* 值的那一行已经满了，那么就再用一次 CSA 操作把当前的 *twos* 和这一行中已有的那两个 *twos* 值合并起来，把 CSA 输出的新 *twos* 值放在这一行，并且把该行的元素占用量重新设为 1，然后继续往下扫描，寻找能够存放 *fours* 值的那一行，以此类推。

处理完数组中的输入值之后，程序接下来再扫描一遍 $m\times2$ 数组（这次扫描过程应该会很短），对所有存满两个元素的行进行压缩，这样所有的行中就只剩下一个有效值了。最后，程序以字组形式来统计数组每行首个元素的种群计数，直到某一行的首元素是 0 为止。于是，数组总的种群计数就是：

$$\text{pop}(\text{第 0 行})+2\text{pop}(\text{第 1 行})+4\text{pop}(\text{第 2 行})+\ldots$$

只要根据上文所述来选取 m 值，那么 $m\times2$ 数组最后一行的两个元素肯定都是 0，于是循环至此必然会终止。

按照这种新思路编写的程序，只需要执行$\lceil \log_2(n+3)\rceil$次种群计数操作就够了。然而可惜的是，这一算法并不实用，因为对存放中间值的那个数组所执行的加载及存储指令，其个数已经超过了节省下来的种群计数指令。试着写出这段程序码（把它优化一下也不难），你就会发现：如果把循环中的全部指令都算上，那么每处理数组中的 1 个字组大约需要 29 条指令，这比前面提到的那种“笨办法”还要差很多。

表 5.1 总结了图 5.8 所示方案在不同的分组方式下所需的指令数。中间两栏所列的数量不含加载及循环控制指令。第 4 栏给出了算法循环体处理数组中的每一个输入字组所需的总指令数。这个数字根据基本 RISC 指令集计算机的编译器统计而来，此类计算机不支持“索引式载入指令”(indexed load)。

表 5.1 统计数组中每个字组元素的种群计数所需的指令个数

算法	除去加载及循环控制后的指令数		循环所需指令总数（以编译器输出结果为准）
	指令数计算公式	For $n_c=5$，$n_p=15$	
朴素算法	n_p+1	16	21
以2个元素为1组	$(n_c+n_p+1)/2$	10.5	14
以4个元素为1组	$(3n_c+n_p+1)/4$	7.75	10
以8个元素为1组	$(7n_c+n_p+1)/8$	6.38	8
以16个元素为1组	$(15n_c+n_p+1)/16$	5.69	7
以32个元素为1组	$(31n_c+n_p+1)/32$	5.34	6.5
以2^n个元素为1组	$n_c+\frac{n_p-n_c+1}{2_n}$	$5+\frac{11}{2^n}$	—

如果数组长度比较小，那么还有比图5.8更好的办法。比方说，如果数组中只有7个字组元素，那么图5.10中的方案就更为高效[Seal1]。它只需要$4n_c+3n_p+4=69$条指令，也就是每处理1个字组，仅需9.86条指令。如果数组长度为$2k-1$（k是任意正整数），那么也存在与本方案相似的高效算法。若数组长度为15，则此算法需要$11n_c+4n_p+6=121$条指令，每处理1个字组元素，需要8.07条指令。

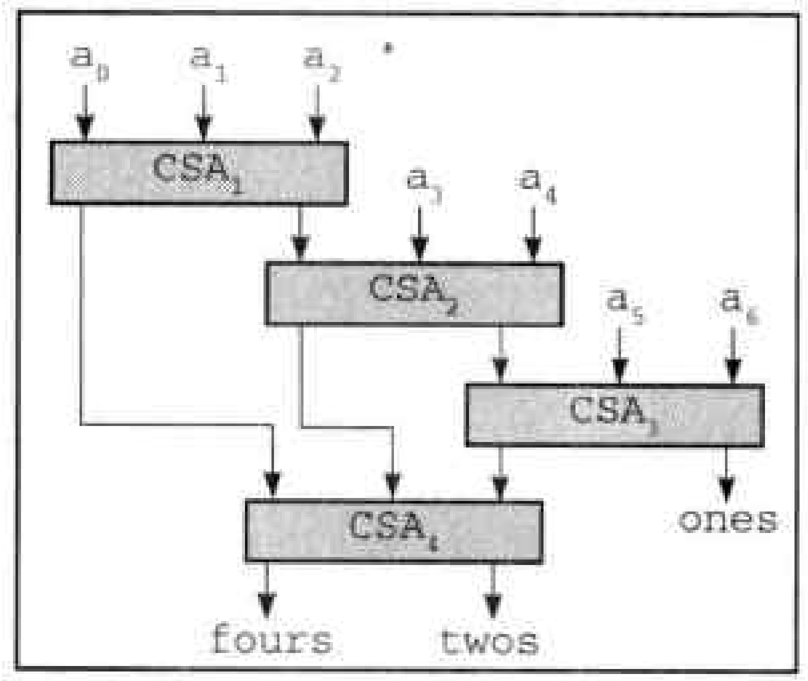


图 5.10 7元素数组的种群计数算法电路图

5.1.4 应用

种群计数的一项应用是计算两个位串的“汉明距离”（Hamming Distance），这是“误差校正码”（error-correcting code）中的一个概念。汉明距离也就是两个位串对应位置上值不相同的位元个数，也就是说：

$$\mathrm{dist}(\boldsymbol{x},\boldsymbol{y})=\mathrm{pop}(\boldsymbol{x}\oplus\boldsymbol{y})$$

此问题可参见[Dewd]一书中谈及“错误校正码”的那一章。

如果数组A的元素分布比较稀疏，那么人们可能会用一种特定的方式将其压缩。种群计数的另外一项应用就是可以用一种相当快的方式直接访问压缩之后的数组。压缩之后的数组只保存已经定义的元素（也就是值非0的元素），它带有一个辅助的位串数组*bits*，其中每个位串都是32位长的字组。若元素$A[i]$有定义，那么整个大位串中，下标为“i”的那个位元就是“1”。为了提升访问速度，还有一个名为*bitsum*的数组，其中每个字组元素*bitsum*[j]的值都是*bits*数组中下标小于j的那些位串元素中值为“1”的位元个数之和。下面演示了一个稀疏数组，其中下标为0、2、32、47、48、95的元素有定义。

bits	*bitsum*	*data*
0x00000005	0	A [0]
0x00018001	2	A [2]
0x80000000	5	A [32]
		A [47]
		A [48]
		A [95]

给定一个下标 i，$0 \leqslant i \leqslant 95$，先在 *bits* 数组所存放的大位串里寻找第 “i” 个位元之前共有多少个值是 “1” 的位元，然后把这个值记为 *sparse_i*，再用它作新的下标来访问压缩之后的数组数据。此计算过程如下所示：

```
j = i >> 5;                    // j = i/32.
k = i & 31;                    // k = rem(i, 32);
mask = 1 << k;                 // A "1" at position k.
if ((bits[j] & mask) == 0) goto no_such_element;
mask = mask - 1;               // 1's to right of k.
sparse_i = bitsum[j] + pop(bits[j] & mask);
```

这种存储方案的开销是：原数组中的每个元素在压缩存储法中都要多用两个位元。

种群计数可以用来生成二项分布随机数（binomially distributed random integer）。BINOMIAL(t,p) 函数能够生成随机整数，其中 t 是尝试次数，$p=1/2$。此函数会生成 t 个随机的二进制位，并计算其中值为 “1” 的位元数。这个算法还可推广至概率 p 不是 1/2 的情况，详情参阅 [Knu2，sec. 3. 4. 1，prob. 27]。

计算机行业一直流传着一个说法：种群计数函数对美国国家安全局非常重要。在 NSA 之外，似乎没人知道其具体用途，不过大致是用在密码工作或海量资料搜寻之类的事情上吧。

5.2 奇偶性

一个位串的 “奇偶性”（parity）是指其中值为 “1” 的位元个数是奇数还是偶数。如果含有奇数个值为 “1” 的位元，那么该位串就是 “奇” 位串，否则就是 “偶” 位串[⊖]。

5.2.1 计算字组的奇偶性

如果字组 $\boldsymbol{x}$ 是 “奇” 的，那么计算结果就是 1，如果是 “偶” 的，则结果为 0。这也就等于把 $\boldsymbol{x}$ 各个位元上的值加起来，再和 2 取模，或者说，对 $\boldsymbol{x}$ 的所有位元做异或操作。

有种办法可以计算奇偶性：先求出 pop（$\boldsymbol{x}$）的值，然后取其最右侧的位元。如果计算机中有 “种群计数指令”，那这样做没问题，不过要是没有的话，可以考虑另外几种办

⊖ 这两个术语的英文分别为 odd parity 与 even parity，又称 “奇数奇偶性” 和 “偶数奇偶性”。——译者注

法，它们比直接用代码实现 pop（x）函数更好。

计算奇偶性较为直接的方法是：

$$y \leftarrow \bigoplus_{i=0}^{n-1} (x \overset{u}{\gg} i)$$

其中 n 是字长，x 的奇偶性就是 y 最右侧的那一位（$\oplus$表示异或操作，不过在本式中，也可以用普通的加法代替）。

如果 n 的值比较大，那么还有一种奇偶性算法比上面那种快得多（演示代码中的 n 是32，移位操作带不带符号都行）：

```
y = x ^ (x >> 1);
y = y ^ (y >> 2);
y = y ^ (y >> 4);
y = y ^ (y >> 8);
y = y ^ (y >>16);
```
(3)

此算法只需 10 个指令，而如果按照前面那种方式来做，就算把循环展开为顺序执行的普通代码，也要花 62 条指令。本算法求出的奇偶性也存放在 y 的最右侧位元中。实际上，不论采用那种算法，只要其中的移位操作是无符号的，那么 y 中第 i 个位元值就表示 x 中此位置及其左方所有位元的奇偶性。此外，由于异或操作的逆运算就是其本身[㊀]，所以 $y_i \oplus y_j$ 就表示 x 中第 $i-1$ 至第 j 位之间这些位元的奇偶性，其中 $i \geqslant j$[㊁]。

这可视为并行计算（parallel computing）中“并行前缀”（parallel prefix）或“扫描”（scan）操作的一个用例［KRS；HS］。假设计算机具备一定数量的处理器，那么就可以把一个看上去时间复杂度[㊂]为 $O(n)$ 的串行算法改为一个复杂度为 $O(\log_2 n)$ 的并行算法。举例来说，如果想对一个数组中所有字组元素的每个位元执行异或操作，那么可以先将算法（3）的 x 视为整个数组，将 $x \gg 1$ 与 $x \gg 2$ 等操作分别视为 $x \gg 32$ 与 $x \gg 64$，然后以字组为单位求异或。这样做需要的（字组长度）异或操作多于简单的从左至右算法，因此在单处理器的电脑中效果并不好。然而在一台处理器数量足够多的并行计算机上，它可以将算法的时间复杂度从 $O(n)$ 降至 $O(\log_2 n)$，其中 n 表示数组中的字组元素个数。

算法（3）较为直接的一项应用就是将“格雷码”（Gray code）整数转换为二进制整数（参见 13.1 节）。

如果在算法（3）的代码中改用左移操作，那么整个 x 字组的奇偶性就存放在 y 的最左侧位元中。而 y 的第 i 个位元值则表示 x 中此位置及其右侧所有位元的奇偶性。该算法又叫“并行后缀”操作，因为 y 中的每一位都是 x 中对应位置及其右侧位元的函数。

若是在算法中采用循环移位操作，那么当 x 为奇时，y 的每一位都是“1”，而 x 为偶时，y 的每一位都是“0”。

㊀ 也就是说，若 x 异或某数 k，结果为 y，则反过来用 y 异或 k，其结果必为 x。——译者注

㊁ 由于 $x_j \oplus x_{j+1} \oplus \ldots \oplus x_{i-1} \oplus x_i \oplus x_{i+1} \oplus \ldots \oplus x_{n-1} = y_j$，而 $x_i \oplus x_{i+1} \oplus \ldots \oplus x_{n-1} = y_i$，所以 $x_j \oplus x_{j+1} \oplus \ldots \oplus x_{i-1} \oplus y_i = y_j$。又因为异或操作是其自身的逆运算，所以 $x_j \oplus x_{j+1} \oplus \ldots \oplus x_{i-1} = y_j \oplus y_i = y_i \oplus y_j$。——译者注

㊂ 算法时间复杂度及“大 O 符号”的含义，请分别参见：http：//zh. wikipedia. org/wiki/算法 # . E6. 97. B6. E9. 97. B4. E5. A4. 8D. E6. 9D. 82. E5. BA. A6 及 http：//zh. wikipedia. org/wiki/大 O 符号。——译者注

在算法（3）中，这 5 行代码可按任意次序排列，只要保证第 1 次运算是针对 x 就行。如果将其反向排列，并且只关心 y 的最低位所存放的奇偶性判断结果，那么最后两行：

```
y = y ^ (y >> 2);
y = y ^ (y >> 1);
```

可以改为［Huef］：

```
y = 0x6996 >> (y & 0xF);
```

这是一种“寄存器内查表操作”[一]。在基本 RISC 架构的计算机上，该算法可以节省 1 条指令，如果不计常数加载所需的指令，那么就节省两条。y 的最低位就是 x 的奇偶性，而 y 的其他位元则不再包含有意义的值了。

下面这种算法需要 9 条指令，运算结果以 1、0 来表示 *x* 的奇偶性（其中移位操作是无符号的）。

```
x = x ^ (x >> 1);
x = (x ^ (x >> 2)) & 0x11111111;
x = x*0x11111111;
p = (x >> 28) & 1;
```

在执行完前两行代码后，*x* 中每个十六进制位[二]的值不是 0 就是 1，具体取值依它所表示的 4 个二进制位而定。乘法操作会把这些十六进制位都加起来，并且存放在权重最高的那个十六进制位中。在把每个十六进制位上的小乘积从上至下累加起来的过程中是不会产生溢出的，因为每一栏最大值是 8，未超过十六进制数所能表示的最大值 15。

可以对 *x* 和 15 求模，以此取代乘法及移位操作。在支持“模除立即数指令”的计算机上，这种（比较慢的）办法只需 8 条指令，比上述算法节省了 1 条指令。

在 64 位机上，需要修改代码（把十六进制常量的位数由 8 扩展至 16，并且把最终的移位量从 28 改为 60），上述算法中的乘法操作才能得出正确结果。在这种情况下，除去权重最高的那一列之外，将其他各列上以 4 个二进制位表示的部分乘积累加起来，其和最大为 15，并未溢出，所以不会影响权重最高的那一列。但是另一方面，这个适用于 64 位计算机的变种算法不可以用“和 15 模除”的办法来取代乘法及移位操作，因为这样算出来的结果是各个十六进制位的和除以 15 之后的余数，而各个十六进制位上的和的最大值可能是 16。

5.2.2 将表示奇偶性的位元添加到 7 位量中

［HAK］的 Item167 中讲了一个很新颖的表达式，对于一个以右对齐方式存放在寄存器里的 7 位量，该表达式可以把其转化为一个具备“偶数奇偶性”的 8 位量。就是说，表达式会在原有的 7 位量左侧再添加 1 位，使整个 8 位量具有“偶数奇偶性”。这行代码本

[一] in-register table lookup，寄存器内存放的常数 0x6996，其二进制表示为 0110 1001 1001 0110，而 *y* & 0xF 的取值只可能在 0 至 15 之间。所以，可将 0x6996 右移 *y* & 0xF 位，据此查得 *x* 的奇偶性。故曰“寄存器内查表法”。——译者注

[二] each hex digit，也就是 4 个二进制位。——译者注

来用于36位计算机，但是在32位机上同样适用。

$$\text{modu}((x\ *\ \mathbf{0x10204081})\ \&\ \mathbf{0x888888FF}, \mathbf{1920})$$

其中，modu（a，b）表示 a 除以 b 的余数，该函数的参数和结果都是无符号的整数。"*"表示将乘法结果与 2^{32} 取模，常数1920是 $15 \cdot 2^7$ 的值。实际上，这个式子是把 x 中每一位的和求出来，然后将其放在原有的7位量左侧，以构成新的 x 值。比方说，原有7位量是 **0x0000 007F**，则计算后的值就是 **0x0000 03FF**，原有7位量是 **0x0000 0055**，则计算后的值就是 **0x0000 0255**。

[HAK] 中还有一个巧妙的公式，可以把7位整数变成具备"奇数奇偶性"的8位整数：

$$\text{modu}((x * \mathbf{0x00204081})\ |\mathbf{0x3DB6DB00}, \mathbf{1152})$$

式子中的1152等于 $9 \cdot 2^7$。为了便于理解此式，我们需要知道8的幂模9之后，结果为±1。如果把式中的 ***0x3DB6 DB*00** 换为 **0xBDB6 DB00**，那么公式算出来的8位值就具备"偶数奇偶性"了。

这些方法在当前的计算机中并不实用，因为内存的价格越来越便宜了，而除法运算依旧很慢。大多数程序员宁可使用简单的查表法来实现上面这些函数。

5.2.3 应用

奇偶性操作广泛应用于计算向数据中添加的校验位。它在 GF(2)[⊖] 的位矩阵乘法中也很有用（此领域的加法操作相当于"异或"）。

5.3 前导0计数

有很多利用二分搜索技术实现的简单算法都可用于计算前导0的个数。下面这种算法模型有若干种变体。在基本RISC架构的计算机上，它需要20至29条指令。其中的比较操作是"逻辑比较"，也就是说，其操作数为无符号整数。

```
if (x == 0) return(32);
n = 0;
if (x <= 0x0000FFFF) {n = n +16; x = x <<16;}
if (x <= 0x00FFFFFF) {n = n + 8; x = x << 8;}
if (x <= 0x0FFFFFFF) {n = n + 4; x = x << 4;}
if (x <= 0x3FFFFFFF) {n = n + 2; x = x << 2;}
if (x <= 0x7FFFFFFF) {n = n + 1;}
return n;
```

此算法的一种变体是用"与"操作取代比较操作：

```
if ((x & 0xFFFF0000) == 0) {n = n +16; x = x <<16;}
if ((x & 0xFF000000) == 0) {n = n + 8; x = x << 8}
...
```

还有一个版本，是用右移位指令来实现的，这样就可以避免载入过大的常量了。

⊖ 两个元素的"伽罗瓦域"（Galois Field），详情参见：http：//en. wikipedia. org/wiki/GF (2)：。——译者注

最后一条 if 语句，只是在最高位为 0 的情况下再加 1，所以可将其改写为如下形式，这样就能减少一个分支：

```
n = n + 1 - (x >> 31);
```

如果把 n 的初始值从 0 改为 1，那么这个赋值语句中的“加 1”操作就可以省略了。基于上文所述，现将该算法代码写为图 5.11，此程序在基本 RISC 架构的计算机上需要 12 至 20 条指令。这个算法还可以继续优化：将 x 以“1”开头的情况与对特殊值 0 的处理结合为一条语句：

```
if ((int)x <= 0) return (-x >> 26) & 32;
```

```
int nlz(unsigned x) {
   int n;

   if (x == 0) return(32);
   n = 1;
   if ((x >> 16) == 0) {n = n +16; x = x <<16;}
   if ((x >> 24) == 0) {n = n + 8; x = x << 8;}
   if ((x >> 28) == 0) {n = n + 4; x = x << 4;}
   if ((x >> 30) == 0) {n = n + 2; x = x << 2;}
   n = n - (x >> 31);
   return n;
}
```

图 5.11　用二分搜索法计算前导 0 的个数

图 5.12 演示了上述算法的逆算法。在前导 0 的数量变多时，它需要的指令数会更少，并且避免了大常量和大的移位量。在基本 RISC 指令集的计算机上，该算法需要 12 至 20 条指令。

还可以用查表法为此算法提速。将最后 4 行可执行代码换成：

```
static char table[256] = {0,1,2,2,3,3,3,3,4,4,...,8);
return n - table[x];
```

很多算法都可以得益于查表法，本书通常不会特意强调这一点。

为了简洁起见，本算法与本节前面讲的那种算法都可以改写为循环形式的代码。例如，图 5.12 中的算法可用循环改写为图 5.13。改写后的代码需要 23 至 33 条基本 RISC 指令，其中 10 条是条件分支指令。

```
int nlz(unsigned x) {
   unsigned y;
   int n;

   n = 32;
   y = x >>16;  if (y != 0) {n = n -16;  x = y;}
   y = x >> 8;  if (y != 0) {n = n - 8;  x = y;}
   y = x >> 4;  if (y != 0) {n = n - 4;  x = y;}
   y = x >> 2;  if (y != 0) {n = n - 2;  x = y;}
   y = x >> 1;  if (y != 0) return n - 2;
   return n - x;
}
```

图 5.12　用二分搜索法递减计算前导 0 的个数

```
int nlz(unsigned x) {
   unsigned y;
   int n, c;

   n = 32;
   c = 16;
   do {
      y = x >> c; if (y != 0) {n = n - c; x = y;}
      c = c >> 1;
   } while (c != 0);
   return n - x;
}
```

图 5.13　利用二分搜索法，以循环形式计算前导 0 的个数

当然了，我们也可以一次只左移1位，然后计数，直至符号位为1，或是一次只右移1位，然后计数，直至整个字组都变成0。这两种算法都很简洁。前者在前导0的个数不多时效果较好，而后者则在前导0的个数很多时效果较好。图5.14结合了这两种算法。在日常应用中，我们经常会将两种算法合并起来使用，然后选择先停下的那个算法，并返回其运算结果。正因为此，所以在这里要专门提一下这种思路。这样写出来的算法在“超标量计算机”（superscalar machine）中运行得很快，因为此类计算机很有可能会并发执行相互独立的指令。（只要指令间各自独立，那么此类计算机就可以同时执行两条或更多条指令。）

```
int nlz(int x) {
   int y, n;

   n = 0;
   y = x;
L: if (x < 0) return n;
   if (y == 0) return 32 - n;
   n = n + 1;
   x = x << 1;
   y = y >> 1;
   goto L;
}
```

图5.14 从左右两端同时计算前导0的个数

在基本RISC指令集的计算机上，该算法所需指令数为 $\min(3+6\mathrm{nlz}(\boldsymbol{x}),5+6(32-\mathrm{nlz}(\boldsymbol{x})))$，在最坏的情况下，需要执行99条指令。我们可以假想一下，如果移位操作以副产品的形式提供了比较操作的结果，那么某台超标量计算机有可能只花1个周期就能执行完整个循环体。即使移位操作不能连带提供比较结果，加上分支开销也只需要两个周期。

可以将图5.11或图5.12中的程序改写为与之对应的无分支代码。图5.15所示无分支代码需要28条基本RISC指令。

若是计算机支持种群计数指令，那么有一种计算前导0个数的好方法，如图5.16所示。其中对x赋值的5条语句可以反过来写，而且实际上按照任何次序编写都行。这段无分支代码需要11条指令。就算计算机没有种群计数指令，这种算法依然值得使用。此时可以按照图5.2所示，用21条指令实现种群计数功能，于是整个算法用32条基本RISC指令即可实现。

Robert Harley先生发明了一种计算 $\mathrm{nlz}(\boldsymbol{x})$ 的方法 [Harley]，它与Seal先生计算 $\mathrm{ntz}(\boldsymbol{x})$ 的方法（参见图5.25）很像。Harley的算法是：通过移位及“或”操作，将权重最高且值为“1”的位元向右传播，然后把结果乘以一个特殊的常量，并把乘积和 2^{32} 取模。此时高6位与 $\boldsymbol{x}$ 中前导0的数目存在一一对应的关系。于是，算法通过右移操作及查表法（用“索引式载入指令”来实现），将这个6位元的标识符转译为 $\boldsymbol{x}$ 中前导0的实际个数。该算法的代码写在图5.17中，它共需14条指令，其中有1条用于乘法操作，此外还要用1条“索引式载入指令”。表格table中未使用的元素以u表示。

乘法操作的乘数是 $7\cdot 255^3$，所以乘法操作也可以用下面几行语句替换。使用替代方案需要19条基本指令，再加1条“索引式载入指令”。

```
x = (x << 3) - x;     // Multiply by 7.
x = (x << 8) - x;     // Multiply by 255.
x = (x << 8) - x;     // Again.
x = (x << 8) - x;     // Again.
```

```
int nlz(unsigned x) {
   int y, m, n;

   y = -(x >> 16);       // If left half of x is 0,
   m = (y >> 16) & 16;   // set n = 16. If left half
   n = 16 - m;           // is nonzero, set n = 0 and
   x = x >> m;           // shift x right 16.
                         // Now x is of the form 0000xxxx.
   y = x - 0x100;        // If positions 8-15 are 0,
   m = (y >> 16) & 8;    // add 8 to n and shift x left 8.
   n = n + m;
   x = x << m;

   y = x - 0x1000;       // If positions 12-15 are 0,
   m = (y >> 16) & 4;    // add 4 to n and shift x left 4.
   n = n + m;
   x = x << m;

   y = x - 0x4000;       // If positions 14-15 are 0,
   m = (y >> 16) & 2;    // add 2 to n and shift x left 2.
   n = n + m;
   x = x << m;

   y = x >> 14;          // Set y = 0, 1, 2, or 3.
   m = y & ~(y >> 1);    // Set m = 0, 1, 2, or 2 resp.
   return n + 2 - m;
}
```

图 5.15 用无分支的二分搜索法计算前导 0 的个数

```
int nlz(unsigned x) {
   int pop(unsigned x);

   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >>16);
   return pop(~x);
}
```

图 5.16 将权重最高且值为“1”的位元向右传播，以此计算前导 0 的个数

```
int nlz(unsigned x) {

   static char table[64] =
     {32,31, u,16, u,30, 3, u,  15, u, u, u,29,10, 2, u,
       u, u,12,14,21, u,19, u,   u,28, u,25, u, 9, 1, u,
      17, u, 4, u, u, u,11, u,  13,22,20, u,26, u, u,18,
       5, u, u,23, u,27, u, 6,   u,24, 7, u, 8, u, 0, u};

   x = x | (x >> 1);      // Propagate leftmost
   x = x | (x >> 2);      // 1-bit to the right.
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >>16);
   x = x*0x06EB14F9;      // Multiplier is 7*255**3.
   return table[x >> 26];
}
```

图 5.17 用 Harley 算法统计前导 0 的个数

有很多乘数都具备这种将一系列数字一一映射到某个区间的独特属性，它们的共同特点是因子均为 $2^k \pm 1$。若表格长度为 64 或 128，则具备该特质的最小数是 0x045B CED1 = 17 · 65 · 129 · 513，此时找不到只有 3 个因子的特殊数字。然而如果表格长度是 256 的话，那么可选的乘数就很多，其中最小的一个为 0x0103 3CBF = 65 · 255 · 1025（使用该方法将节省两条指令，然而代价是要准备一张更大的表格）。

Julius Goryavsky 先生提出了若干种 Harley 算法的变体 [Gor]，其中有的方案多花了几条指令来减少表格大小，有的方案改进了并发执行能力，有的方案则引入了其他一些良好特性。图 5.18 演示了其中一种变形算法，如果用移位和加法指令来实现其中的乘法操作，那么它将大大胜过前述算法。与图 5.17 相比，该算法只是改动了表格数

据以及右移16位之后做乘法的那两行代码。算法中的乘数可以分解为511·2047·16383（再与2^{32}取模），因而实现乘法只需6条指令，而不用像刚才的算法那样要用8条。如果计算机有“and not”指令，那么总共能省两条指令，若是不支持“and not”，那么能省1条。

```
...
static char table[64] =
  {32,20,19, u, u,18, u, 7,  10,17, u, u,14, u, 6, u,
    u, 9, u,16, u, u, 1,26,   u,13, u, u,24, 5, u, u,
    u,21, u, 8,11, u,15, u,   u, u, u, 2,27, 0,25, u,
   22, u,12, u, u, 3,28, u,  23, u, 4,29, u, u,30,31};
...
x = x & ~(x >> 16);
x = x*0xFD7049FF;
...
```

图5.18 用Goryavsky改写的Harley算法统计前导0的个数

5.3.1 浮点数算法

“浮点数后正规化功能”（floating-point post-normalization facility）可以用来计算前导0的个数，在使用IEEE格式的浮点数[㊀]时，此算法效果很好。其思路是，将给定的无符号整数转换为双精度浮点数，然后提取出其中的指数信息，再将该指数从一个定值中减去。图5.19列出了此算法的完整代码。

这段代码使用了C++语言的“匿名联合”数据结构（anonymous union），以便把整数与双精度浮点数重叠放置在同一个内存区域中。如果计算机以“小端序”的方式存储数据，那么变量LE的值必须为1，若以“大端序”方式存储，则LE必须为0。为了让算法在k=0时也能正常执行，我们需要给该值加上一个0.5或与之类似的小数量。

此处不打算分析这段代码的执行时间，因为各种计算机的浮点数运算能力差别很大。举例来说，很多计算机的浮点数寄存器与整数寄存器是分开放置的，在这种计算机里，如果要通过内存传输数据，那么必须把整数转换为浮点数，然后将运算结果移动到整数寄存器中。

```
int nlz(unsigned k) {
   union {
      unsigned asInt[2];
      double asDouble;
   };
   int n;

   asDouble = (double)k + 0.5;
   n = 1054 - (asInt[LE] >> 20);
   return n;
}
```

图5.19 利用IEEE格式的浮点数求出前导0的个数

图5.19中的代码不符合C或C++语言的ANSI标准，因为它将同一块内存用作两种数据类型。因此，我们不能保证它能在某个特定机型上正常运行，也不能保证某种特定的编译器是否能正确编译这样的代码。在运行AIX操作系统[㊁]的计算机上使用IBM的XLC编译

㊀ 浮点数格式及其“指数”、“正规化”等问题，请参考：http：//zh.wikipedia.org/wiki/双精度浮点数。——译者注

㊁ AIX是Advanced Interactive eXecutive的缩写，这是IBM专有的UNIX操作系统的商标名。该系统可运行于POWER及PowerPC等架构的计算机上。详情参见：http：//zh.wikipedia.org/wiki/AIX。——译者注

器[⊖]是可以的，在 AIX 系统、Windows 2000 及 XP 系统上使用 GCC 编译器，不论进行何种级别的优化，都能正常编译此代码（至少在笔者编写本书时如此）。如果改用如下代码来实现内存叠加定义（overlay defining）的话：

```
xx = (double)k + 0.5;
n = 1054 - (*((unsigned *)&xx + LE) >> 20);
```

那么在这些操作系统上编译时，就不能打开优化选项了。而且这段代码一不小心又违反了 ANSI 标准的另外一条规则，那就是：只可在指向数组的指针上做算数运算 [Cohen]。不过这段代码编译失败的原因还是在于其违反了前面那条不得叠加定义的规则。

尽管该算法的代码颇为古怪，让人看上去不放心[⊜]，不过此处还是给出它的 3 种变体。

```
asDouble = (double)k;
n = 1054 - (asInt[LE] >> 20);
n = (n & 31) + (n >> 9);

k = k & ~(k >> 1);
asFloat = (float)k + 0.5f;
n = 158 - (asInt >> 23);

k = k & ~(k >> 1);
asFloat = (float)k;
n = 158 - (asInt >> 23);
n = (n & 31) + (n >> 6);
```

在第 1 种变体算法中，`k = 0` 的特殊情况不是通过预先对其值加上 0.5 来解决的，而是对算出来的 `n` 再次执行整数算术（如果 `k = 0`，那么第 2 步算出来的 `n` 在未修正的情况下就是 `1054`，写成十六进制就是 `0x41E`）。

剩下两种变体算法使用单精度浮点数，此时显然也要相应地修改图 5.19 中的“匿名联合”数据结构。这两种算法都会产生一个新的问题：如果舍入模式是“向最近的浮点数舍入”（通常使用此方式）或是“向正无穷”进位，那么我们想求的结果可能会在舍入过程中被丢弃。如果用的是“向最近的浮点数舍入”，那么当 k 位于十六进制数 FFFF FF80 至 FFFF FFFF，7FFF FFC0 至 7FFF FFFF，3FFF FFE0 至 3FFF FFFF 等范围时，就会发生舍入问题。在舍入过程中，加 1 操作会持续向左传播进位，从而导致原来权重最高且值为“1”的那个位元发生改变。为此我们要提前把该位元右侧的那个位元置 0，以便阻断可能发生的持续进位现象。如果 k 是 64 位量，那么图 5.19 中的代码及 3 个变种算法里的第 1 种算法也要据此修正。

GNU C/C++编译器有一种特性，可以将上面任何一种算法预定义为一个宏，这样本来需要调用这些算法函数的地方就会直接展开成内联代码了 [Stall]。基于此特性，通过调用宏来计算表达式的值时，其对应程序语句连同声明就会被一并插入当前代码中。宏里面的语句序列通常以表达式结尾，该表达式就充当整个语句序列的值。下面列出了这样一个宏定义，它对应于采用单精度浮点数的那种算法变体。（在 C 语言中，习惯上用

⊖ 由 IBM 开发的专属编辑器，详情参见：http://www.ibm.com/software/products/us/en/ccompfami。——译者注

⊜ 这段代码之所以让人觉得容易出错（flaky），是因为我们是在用 C 语言来编写它。如果改用机器语言，或是让某种针对特定机型的编译器生成机器代码，那么就完全不必担心了。

大写字母做宏的名称。）

```
#define NLZ(kp) \
   ({union {unsigned _asInt; float _asFloat;}; \
     unsigned _k = (kp), _kk = _k & ~(_k >> 1); \
     _asFloat = (float)_kk + 0.5f; \
     158 - (_asInt >> 23);})
```

使用下划线是为了避免变量名和宏展开之后的参数“kp”冲突，因为编码者在命名变量时，通常不会以下划线开头。

5.3.2 比较两个字组前导 0 的个数

有一种简单的办法可以在不计算 nlz(x) 与 nlz(y) 的前提下，判断出 x 与 y 谁的前导 0 个数比较多［Knu5］。下面列出了 3 组互为充要条件的等价命题，剩下 3 种没有列出的关系式显然可以通过反转右方那一栏中的不等号方向而得出。

$$\text{nlz}(\boldsymbol{x})=\text{nlx}(\boldsymbol{y})\text{当且仅当}(\boldsymbol{x}\oplus\boldsymbol{y})\overset{u}{\leqslant}(\boldsymbol{x}\,\&\,\boldsymbol{y})$$

$$\text{nlz}(\boldsymbol{x})<\text{nlx}(\boldsymbol{y})\text{当且仅当}(\boldsymbol{x}\,\&\,\neg\,\boldsymbol{y})\overset{u}{>}\boldsymbol{y}$$

$$\text{nlz}(\boldsymbol{x})\leqslant\text{nlx}(\boldsymbol{y})\text{当且仅当}(\boldsymbol{y}\,\&\,\neg\,\boldsymbol{x})\overset{u}{\leqslant}\boldsymbol{x}$$

5.3.3 与对数函数的关系

“nlz”函数本质上就是以 2 为底的整数对数函数。对于非 **0** 的无符号数 x 来说：

$$\lfloor\log_2(\boldsymbol{x})\rfloor=31-\text{nlz}(\boldsymbol{x}),\text{且}$$
$$\lceil\log_2(\boldsymbol{x})\rceil=32-\text{nlz}(\boldsymbol{x-1})$$

此问题请参见本书 11.4 节。

另一个与之相关的函数是 bitsize，也就是将参数以 2 补码形式保存所需的位元个数。该函数定义如下：

$$\text{bitsize}\ (x)\ =\begin{cases}1, & x=-1\text{ 或 }0,\\ 2, & x=-2\text{ 或 }1,\\ 3, & -4\leqslant x\leqslant-3\text{ 或 }2\leqslant x\leqslant 3,\\ 4, & -8\leqslant x\leqslant-5\text{ 或 }4\leqslant x\leqslant 7,\\ \cdots & \cdots\\ 32, & -2^{31}\leqslant x\leqslant-2^{30}+1\text{ 或 }2^{30}\leqslant x\leqslant 2^{31}-1.\end{cases}$$

由函数定义可知，bitsize（x）＝bitsize（$-x-1$），而 $\boldsymbol{-x-1=\neg x}$，所以 bitsize 函数的一种计算方式如下（其中移位操作是带符号的）：

```
x = x ^ (x >> 31);            // If (x < 0) x = -x - 1;
return 33 - nlz(x);
```

还有一种算法，也能计算出 bitsize（x）的函数值，只不过在 $x=0$ 时，该算法的结果也是 0。

```
32 - nlz(x ^ (x << 1))
```

5.3.4 应用

前导 0 计数函数的两个最重要的用途是模拟浮点数的算术操作及参与各种除法算法(分别参见图 9.1 及图 9.3)。除此以外，该指令似乎还有其他一些用途。

利用该函数，仅需 3 条指令就可算出比较谓词“$\boldsymbol{x}=\boldsymbol{y}$”的值（参见 2.12 节），而且它还能帮我们计算一些初等函数（分别参见 11.1.1、11.1.2、11.3.2、11.4.2 节）。

有一个比较新潮的应用，就是利用该函数生成按指数分布的随机整数：先生成均匀分布的随机整数，然后对其结果做 nlz 运算 [GLS1]。这种算法生成 0 的概率是 1/2，生成 1 的概率是 1/4，生成 2 的概率是 1/8，以此类推。此外还有一种用法，就是在字组中帮助查找连续出现且长度为某个定值的一串“1”位元（或一串“0”位元），这在某些“磁盘块分配算法”（disk block allocation algorithm）中会用到。对于后两种应用来说，也可以使用“后缀 0 计数”函数。

5.4 后缀 0 计数

如果可以使用前导 0 计数指令，那么计算后缀 0 个数的最佳算法也许就是将其转化为前导 0 计数问题：

$$\mathbf{32}-\mathrm{nlz}(\neg\boldsymbol{x}\ \&(\boldsymbol{x}-\mathbf{1}))$$

如果种群计数指令可用，那么还有个稍微好一点的算法，就是构造一个与后缀 0 个数相同的掩码，然后统计其种群计数 [Hay2]。该算法的两种算式如下：

$$\mathrm{pop}(\neg\boldsymbol{x}\ \&(\boldsymbol{x}-\mathbf{1}))$$

$$\mathbf{32}-\mathrm{pop}(\boldsymbol{x}\,|-\boldsymbol{x})$$

使用诸如 2.1 节中列出的那些方法，构造出一个标识 $\boldsymbol{x}$ 中后缀 0 数目的掩码，这样就可以写出该算法的很多种变形版本了。这些算法也适用于那些不支持位计数指令的计算机。如果用图 5.2 中的代码来实现 pop($\boldsymbol{x}$) 函数，那么上述第 1 个表达式大约需要 3+21=24 条无分支指令。

图 5.20 演示了一种直接求值的算法，在 x 不为 0 时，大约需要 12 至 20 条指令。

```
int ntz(unsigned x) {
   int n;

   if (x == 0) return(32);
   n = 1;
   if ((x & 0x0000FFFF) == 0) {n = n +16; x = x >>16;}
   if ((x & 0x000000FF) == 0) {n = n + 8; x = x >> 8;}
   if ((x & 0x0000000F) == 0) {n = n + 4; x = x >> 4;}
   if ((x & 0x00000003) == 0) {n = n + 2; x = x >> 2;}
   return n - (x & 1);
}
```

图 5.20 用二分搜索法求后缀 0 的个数

若是编译器不够聪明，没有在值得优化时自动优化代码，那么你可以直接把 n + 16 简化为 17（这并不影响我们统计出来的该算法所需的指令数）。

图 5.21 列出的这种算法所用的常量比刚才那种更小，而且操作符也更为简单。它需要 12 至 21 条基本 RISC 指令。与上述算法不同，如果后缀 0 的数目比较小，那么图 5.21 中的代码所执行的指令数会比较多，不过它一直往下执行的次数[⊖]也比较多。

```
int ntz(unsigned x) {
   unsigned y;
   int n;

   if (x == 0) return 32;
   n = 31;
   y = x <<16;  if (y != 0) {n = n -16;  x = y;}
   y = x << 8;  if (y != 0) {n = n - 8;  x = y;}
   y = x << 4;  if (y != 0) {n = n - 4;  x = y;}
   y = x << 2;  if (y != 0) {n = n - 2;  x = y;}
   y = x << 1;  if (y != 0) {n = n - 1;}
   return n;
}
```

图 5.21　用较小常量计算后缀 0 数目的算法

return 语句上方的那行代码也可改写为：

```
n = n - ((x << 1) >> 31);
```

这样会减少 1 个分支，但是所需指令总数不变。

如果从执行的指令数来看，上述算法很难胜过“搜索树”算法［Aus2］。图 5.22 演示了适用于 8 位参数的该算法。这段程序一般需要执行 7 条指令，只是最后两种情况（当需要返回的值是 7 和 8 时）需要 9 条。32 位版本要执行 11 至 13 条指令。然而不巧的是，当字长变大时，程序的长度也增长得很快。刚说的那种 8 位版本需要 12 行可执行源代码，编译成指令后约是 41 条，而 32 位版本需要 48 行代码，编译出来的指令大约为 164 条，64 位版本的代码行数与指令个数都是 32 位的两倍。

如果后缀 0 的个数比较少（或比较多），那么用图 5.23 所示的简单循环算法会非常快。左右两侧算法所需执行的基本 RISC 指令数分别为 $5+3\text{ntz}(x)$、$3+3(32-\text{ntz}(x))$。

```
int ntz(char x) {
   if (x & 15) {
      if (x & 3) {
         if (x & 1) return 0;
         else return 1;
      }
      else if (x & 4) return 2;
      else return 3;
   }
   else if (x & 0x30) {
      if (x & 0x10) return 4;
      else return 5;
   }
   else if (x & 0x40) return 6;
   else if (x) return 7;
   else return 8;
}
```

图 5.22　用“二叉搜索树”算法求后缀 0 的个数

```
int ntz(unsigned x) {
   int n;

   x = ~x & (x - 1);
   n = 0;                      // n = 32;
   while (x != 0) {            // while (x != 0) {
      n = n + 1;               //    n = n - 1;
      x = x >> 1;              //    x = x + x;
   }                           // }
   return n;                   // return n;
}
```

图 5.23　用简单的计数循环求后缀 0 的个数

⊖ 原文为 fall—through branch，大意为“跌落分支”、“下坠分支”、“落空分支”，指的是某种大分支体系中的一系列特定小分支。程序在执行完一系列小分支语句中的某一条后，并不立刻跳出整个大分支体系，而是继续向下执行其他小分支。通常用于指 switch 语句中的各个 case 分支。详情参见：http：//en.wikipedia.org/wiki/Switch _ statement。——译者注

Dean Gaudet 先生发明了一种有趣的算法［Gaud］，如果选取恰当的指令，那么就可以实现出无分支、无加载指令（也就是不使用查表法）而且并发执行度较高的代码。该算法如图 5.24 所示。

```
int ntz(unsigned x) {
   unsigned y, bz, b4, b3, b2, b1, b0;

   y = x & -x;                 // Isolate rightmost 1-bit.
   bz = y ? 0 : 1;             // 1 if y = 0.
   b4 = (y & 0x0000FFFF) ? 0 : 16;
   b3 = (y & 0x00FF00FF) ? 0 : 8;
   b2 = (y & 0x0F0F0F0F) ? 0 : 4;
   b1 = (y & 0x33333333) ? 0 : 2;
   b0 = (y & 0x55555555) ? 0 : 1;
   return bz + b4 + b3 + b2 + b1 + b0;
}
```

图 5.24　用 Gaudent 算法求后缀 0 的个数

正如大家所见，代码中有 6 个地方使用了 C 语言的“条件表达式”（conditional expression）。这种表达式的语法形式是 a? b：c，如果 a 是“**真**”（非零值），那么表达式的值就是 b，如果 a 是“**假**”（零值），那么表达式的值就是 c。虽说条件表达式一般都会编译成比较及分支指令，但是对于图 5.24 中这种简单的情况来说，若是计算机支持“与 0 比较指令”（compare for equality to zero instruction），能够在操作数为 0 时将寄存器设为 1，在操作数非零时将寄存器设为 0，那么不需要分支指令即可实现。假如计算机支持“条件移动指令”，那么实现起来也不需要分支。如果用比较指令来实现条件表达式，那么向 b3 赋值的那条语句共需 5 条基本 RISC 指令：两条用于生成十六进制常数，一条“与指令”，一条“比较指令”，再加一条移位长度为 3 的左移指令。（实现第 1、第 2 及最后一个条件表达式分别需要 1、3、4 条指令。）

这段代码编译完毕后一共是 30 条指令。带有条件表达式的 6 行代码可以并发执行。在一台并发执行能力足够高的计算机上，此算法用 10 个周期即可执行完。当前的计算机通常没有如此高的并发执行能力，所以从实际角度出发，可以将前两处用到 y 的地方均改为 x，这样前 3 行可执行代码就能并发执行了。

David Seal 先生发明了一种计算 ntz（x）的办法［Seal2］，其思路是，将 x 值的 2^{32} 种可能性压缩到一个稠密集合（dense set）中，然后查表。他先用表达式 x & $-x$ 将可能出现的取值压缩到一个较小的范围内。这个表达式的值只有 1 个位元是“1”，它的位置与 x 中最右侧的“1”相同。如果 x 是 0，则该表达式的值也是 0。这样的话，x & $-x$ 就只可能取 33 个值了。然而这并非稠密集合，因为这 33 个值位于从 0 到 2^{31} 这个相当大的范围内。

现在需要构建一个稠密集合，使得其中的 33 个整数与 x & $-x$ 的 33 种取值一一对应。Seal 找到了一个特定的常数，将该数乘以 x & $-x$ 之后，在乘积的低权重字组中，高 6 位和 x & $-x$ 有一一对应关系。因为 x & $-x$ 不是 2 的整数幂就是 0，所以乘法操作要么相当于常量向左移位操作，要么相当于乘以一个值是 0 的数。只用权重最高的 5 位是不够

的，因为要将 $x \& -x$ 映射成 33 个互不相同的值。

图 5.25 列出了这段算法，其中未使用的表格元素均以 u 标出。

```
int ntz(unsigned x) {

   static char table[64] =
     {32, 0, 1,12, 2, 6, u,13,   3, u, 7, u, u, u, u,14,
      10, 4, u, u, 8, u, u,25,   u, u, u, u, u,21,27,15,
      31,11, 5, u, u, u, u, u,   9, u, u,24, u, u,20,26,
      30, u, u, u, u,23, u,19,  29, u,22,18,28,17,16, u};

   x = (x & -x)*0x0450FBAF;
   return table[x >> 26];
}
```

图 5.25 Seal 算法，计算后缀 0 的数量

举例来说，如果 x 是 16 的奇数倍，那么 x & - x = 16，于是乘法操作就相当于把常数左移 4 位。在乘积的低权重字组中，高 6 位的值是 01 0001，也就是十进制数 17。表格里下标为 17 的元素其值是 4。这个值是正确的，因为 16 的奇数倍其后缀 0 的个数必然是 4。

有成千上万个常数都能产生这种映射效果，其中最小的一个是 0x0431 472F，最大的是 0xFDE7 5C6D。Seal 选择的这个常数的好处是：只用很少的几条移位及加法指令就能实现出来。由于 0x0450 FBAF=17 • 65 • 65 535，所以乘法操作可以用下列代码替换：

```
x = (x << 4) + x;     // x = x*17.
x = (x << 6) + x;     // x = x*65.
x = (x << 16) - x;    // x = x*65535.
```

用上述代码替换后，图 5.25 只需 9 条基本指令再加 1 条“索引式载入指令”即可实现。Seal 对 ARM 指令集很感兴趣，在此种指令集中，移位与加法操作仅需 1 条指令即可，所以在 ARM 架构的计算机中，实现这段代码只需 6 条指令，其中包括了 1 条“索引式载入指令”。

为了让乘法操作更容易用移位及加法操作表示出来，大家可能想找找看有没有 $(2^{k_1}\pm1)(2^{k_2}\pm1)$ 形式的常数，能够起到一一映射的效果。对于长度是 64 的表格来说，没有这样的整数。另外一个可用的整数也要分解成 3 个因子：0x08A1 FBAF=17 • 65 • 131 071。把表长增加到 128 或 256 也不行。然而，如果表长是 512，那么就能找到适合 $(2^{k_1}\pm1)$ $(2^{k_2}\pm1)$ 形式的整数了，其中值最小的一个是 0x0080 FF7F=129 • 65 535。此时表中所应填写的数值留待各位读者来完成。

Seal 算法还有一种基于德布鲁因循环序列的变体 [LPR]。长度为固定值的各种字母排列组合都能在循环序列中找到，而且只出现 1 次。举例来说，如果字母表里有 3 个字母 $\{a, b, c\}$，那么由这 3 个字母中任意两个排列而成的各种子序列在 *aabacbbcc* 这个德布鲁因序列中都能找到。请注意，*ca* 子序列是由大序列中最后一个字母绕回到头一个字母而构成的。若字母表长度为 k，而子序列长度为 n，那么就有 k^n 个这样的子序列。如果一个循环序列想包含全部子序列，那么它的长度至少是 k^n，此时从每个位置向右寻找，都能发现一个排列方式不同的子序列。

本例中假设字母表为 {0，1}，子序列长度是 5。为了处理 32 位长的字组，我们需要找到一个循环序列，让它能包含全部 32 种子序列：00000、00001、00010、…、11111。如果一个循环序列的头 4 个位元是 0，那么我们就可以用它来计算 ntz（x）的值了：先按照 Seal 算法，把 x 里权重最低且值为“1”的位元析出，然后通过乘法操作选出德布鲁因循环序列中的 5 个位元。因为这 5 个位元与乘数 x & $-x$ 存在一一对应的关系，所以可根据此映射关系，用查表法求出 x 的后缀 0 的个数。该算法所用的德布鲁因循环序列是：

```
0000 0100 1101 0111 0110 0101 0001 1111.
```

这实际上就是个循环序列，因为在运算过程中，该序列尾部实际上还有很多个“0”没有列在上面所写的 32 个位元中。这样的话，实际效果就等于折回到序列开头㊀。

ntz（x）的值有 33 种，而德布鲁因循环序列的 5 位子序列只有 32 种。因此，必然存在 ntz（x）值不同而查表所得值却相同的情况。发生冲突的是 0 和那些最右侧位元是“1”的数。为了解决这个问题，代码需要先判断 x 是否为 0，如果是 0，就返回 32。若计算机支持“比较谓词指令”，那么将最后一行代码改为下列形式，即可实现无分支算法了：

```
return table[x >> 27] + 32*(x == 0);
```

现在比较一下这两种算法。Seal 的算法不需要检测 x 是否为 0，而且它有可能只用 6 条指令就能实现出来，而德布鲁因算法用的表格更小。图 5.26 中的德布鲁因循环数列是由 Danny Dubé 先生发现的 [Dubé]。这个数字选得好，因为与该数的乘法可以用 8 条基本指令实现出来。常数 0x04D7 651F 可以写成（2047 • 5 • 256＋1）• 31，于是只用移位、加法、减法指令就可以实现乘法了。

```
int ntz(unsigned x) {

   static char table[32] =
     { 0, 1, 2,24, 3,19, 6,25,  22, 4,20,10,16, 7,12,26,
      31,23,18, 5,21, 9,15,11,  30,17, 8,14,29,13,28,27};

   if (x == 0) return 32;
   x = (x & -x)*0x04D7651F;
   return table[x >> 27];
}
```

图 5.26 用德布鲁因循环序列求后缀 0 的个数

John Reiser 先生指出，还有一种方法 [Reiser] 也能把 Seal 算法中 x & －x 的 33 种取值映射到一个稠密集合中互不相同的整数上，那就是：对其做除法，并取余数。具备这种映射能力的最小除数是 37。采用新算法的代码如图 5.27 所示，表格中值为 u 的元素未使用。

㊀ 这里的意思是说，如果 x & $-x$ 的值是 2^{28}、2^{29}、2^{30}、2^{31}，那么乘法操作就相当于将代表此循环序列的常数 0x04D7 651F 左移 28～31 位，在移位结果的前 5 个位元中，就会分别出现 1～4 个后缀 0，而该常数的头 4 个位元是 0，恰好和这 4 种情况下所取的子序列（11110、11100、11000、10000）相等。——译者注

```
int ntz(unsigned x) {

   static char table[37] = {32,  0,  1, 26,  2, 23, 27,
                  u,  3, 16, 24, 30, 28, 11,  u, 13,  4,
                  7, 17,  u, 25, 22, 31, 15, 29, 10, 12,
                  6,  u, 21, 14,  9,  5, 20,  8, 19, 18};

   x = (x & -x)%37;
   return table[x];
}
```

图 5.27　用 Reiser 算法求后缀 0 的个数

有个有趣的现象：如果 x 是均匀分布的，那么其后缀 0 的平均个数就非常接近 1.0。为了说明这一点，我们可以用 p_i 表示后缀 0 为 n_i 的概率，然后把 $p_i n_i$ 的乘积加起来，于是可得：

$$
\begin{aligned}
S &\cong \frac{1}{2}\cdot 0+\frac{1}{4}\cdot 1+\frac{1}{8}\cdot 2+\frac{1}{16}\cdot 3+\frac{1}{32}\cdot 4+\frac{1}{64}\cdot 5+\cdots \\
&\cong \sum_{n=0}^{\infty}\frac{n}{2^{n+1}}
\end{aligned}
$$

为了求出此式的和，我们考虑如下数组：

$$
\begin{array}{cccccc}
1/4 & 1/8 & 1/16 & 1/32 & 1/64 & \cdots \\
 & 1/8 & 1/16 & 1/32 & 1/64 & \cdots \\
 & & 1/16 & 1/32 & 1/64 & \cdots \\
 & & & 1/32 & 1/64 & \cdots \\
 & & & & 1/64 & \cdots \\
 & & & & & \cdots
\end{array}
$$

由于每一列的和就是 S 中的一项，因此 S 的值就是整个数组中全部数字之和。而每一行的和则是：

$$
\begin{aligned}
&1/4+1/8+1/16+1/32+\ldots=1/2 \\
&1/8+1/16+1/32+1/64+\ldots=1/4 \\
&1/16+1/32+1/64+1/128+\ldots=1/8 \\
&\cdots
\end{aligned}
$$

把每一行的和加起来，就是：$1/2+1/4+1/8+\ldots=1$。原级数的绝对收敛性[⊖]保证了重排后的运算结果也是正确的。

有时需要一个与 ntz($\boldsymbol{x}$) 相似的函数，只是要对参数 0 做特殊处理。由于 0 可能代表某种错误，所以此时的函数值要很容易地与正常情况下的取值区分开。举例来说，我们想定义一个“$\boldsymbol{x}$ 中值为 2 的因子个数”（The number of factors of 2 in $\boldsymbol{x}$）函数：

⊖ Absolute Convergence，绝对收敛。按任意次序排列绝对收敛数列的各项不会改变其总和。此概念的含义及其判别方法分别参见：http://zh.wikipedia.org/wiki/绝对收敛与 http://zh.wikipedia.org/wiki/审敛法。——译者注

$$\text{nfact}(\boldsymbol{x}) = \begin{cases} \text{ntz}(\boldsymbol{x}), & \boldsymbol{x} \neq \boldsymbol{0}, \\ -1, & \boldsymbol{x} = \boldsymbol{0}. \end{cases}$$

可以用下面的式子求出函数值：

$$\mathbf{31} - \text{nlz}(\boldsymbol{x} \mathbin{\&} -\boldsymbol{x})$$

应用

[GLS1] 提出了一些后缀 0 计数函数的有趣用法。该函数又叫“标尺函数”（ruler function），因为它可以算出标尺上二等分、四等分、八等分等刻度线的高度。

此函数可以用在 R. W. Gosper 的循环检测算法中。接下来将详述此算法，因为它相当优雅，而且尽管初看上去平凡无奇，但实际上它的用途却比较广泛。

假设 X_0，X_1，X_2，…是由 $X_{n+1}=f\ (X_n)$ 所定义的序列，若 f 的值域有限，则该序列一定具有周期性。也就是说，X_0，X_1，…，$X_{\mu-1}$的后面肯定会重复出现一个值与前面各数相同的 X_μ，$X_{\mu+1}$，…，$X_{\mu+\lambda-1}$，并且一直循环往复（再次重复时，$X_\mu=X_{\mu+\lambda}$，$X_{\mu+1}=X_{\mu+\lambda+1}$，$\lambda$ 为循环周期）。给定函数 f，循环检测问题就是找到首次出现重复数值的下标 μ 以及周期 λ。循环检测技术可以测试随机数生成器，也可以检测链表（linked list）中是否存在循环。

我们当然可以在生成序列值的过程中将其全部保存起来，然后在生成新值时，将它与过去的每一个值相比较，这样立刻就能发现第 2 次循环从何处开始。然而，还有一些算法从空间和时间上都比这个方法的效率高很多。

这其中最简单的要数 R. W. Floyd 给出的算法了 [Knu2, sec. 3.1, prob. 6]。该算法反复执行如下过程：

$$x=f(x)$$
$$y=f(f(y))$$

其中 x 与 y 的初始值都是 X_0。在执行完 n 步之后，$x=X_n$ 而 $y=X_{2n}$。比较这两个值，若相等，则说明 X_n与 X_{2n}之间存在整数倍个 λ 周期，也就是说，$2n-n=n$，而 n 必为λ 的倍数。μ 的值可以这样求：从头开始重新生成序列，先比较 X_0 与 X_n，然后比较 X_1 与 X_{n+1}，以此类推，直至找到 $X_\mu=X_{n+\mu}$为止，此时的下标就是 μ。继续向下生成序列元素，并比较 X_μ与 $X_{\mu+1}$、$X_{\mu+2}$等项是否相等，这样最终就能求出 λ 的值了。该算法需要的空间很少，而且有界，不过它却要计算很多次 f 函数。

Gosper 算法 [HAK, item 132; Knu2, Answers to Exercises for Section 3.1, exercise 7] 可以找到周期λ，但是却不寻找第一次出现重复数值的下标 μ。它的主要特性是绝不会回过头去重新计算 f 函数，而且该算法从时间与空间角度看都很经济。此算法没有空间界限，它需要一个长度为 $\log2(\Lambda)+1$ 的表格，其中 Λ 是周期可能取到的最大值。实际上该算法用不了多少空间，比方说，如果知道 $\Lambda\leqslant2^{32}$，那么 33 个字组就足够了。

图 5.28 用 C 语言描述了 Gosper 的算法。这个用 C 语言写成的函数接受两个参数，一个表示待分析的函数 f，另一个表示起始值 X_0。该函数返回 μ 的下限、上限以及周期

λ。（尽管Gosper算法不能直接算出μ，但是它却能给出下限μ_l与上限μ_u，使得$\mu_u-\mu_l+1\leqslant\max(\lambda-1, 1)$。）此算法的步骤是：在$n=1, 2, \cdots$时，于出现在$X_n$之前的$n$个元素中选出$\lfloor\log_2 n\rfloor+1$个元素构成子集，然后将$X_n$与子集中的各个元素相比较。子集中的首个元素$X_i$是距离$X_n$最近，且下标$i$再加1之后最末端位元值是“1”的那个元素（也就是说，i是n之前的那个偶数）；子集中的第2个元素是最接近X_n且下标i再加1后结尾恰好有1个“0”位元的那个元素；子集中的第3个元素是最接近X_n且下标i再加1后结尾恰好有两个“0”位元的那个元素，以此类推。

```
void ld_Gosper(int (*f)(int), int X0, int *mu_l,
                              int *mu_u, int *lambda) {
   int Xn, k, m, kmax, n, lgl;
   int T[33];

   T[0] = X0;
   Xn = X0;
   for (n = 1; ; n++) {
      Xn = f(Xn);
      kmax = 31 - nlz(n);           // Floor(log2 n).
      for (k = 0; k <= kmax; k++) {
         if (Xn == T[k]) goto L;
      }
      T[ntz(n+1)] = Xn;             // No match.
   }
L:
   // Compute m = max{i | i < n and ntz(i+1) = k}.

   m = ((((n >> k) - 1) | 1) << k) - 1;
   *lambda = n - m;
   lgl = 31 - nlz(*lambda - 1); // Ceil(log2 lambda) - 1.
   *mu_u = m;                   // Upper bound on mu.
   *mu_l = m - max(1, 1 << lgl) + 1;// Lower bound on mu.
}
```

图5.28　Gosper循环检测算法

因此，该算法的比较过程如下：

$X_1: X_0$	$X_7: X_6, X_5, X_3$	$X_{13}: X_{12}, X_9, X_{11}, X_7$
$X_2: X_0, X_1$	$X_8: X_6, X_5, X_3, X_7$	$X_{14}: X_{12}, X_{13}, X_{11}, X_7$
$X_3: X_2, X_1$	$X_9: X_8, X_5, X_3, X_7$	$X_{15}: X_{14}, X_{13}, X_{11}, X_7$
$X_4: X_2, X_1, X_3$	$X_{10}: X_8, X_9, X_3, X_7$	$X_{16}: X_{14}, X_{13}, X_{11}, X_7, X_{15}$
$X_5: X_4, X_1, X_3$	$X_{11}: X_{10}, X_9, X_3, X_7$	$X_{17}: X_{16}, X_{13}, X_{11}, X_7, X_{15}$
$X_6: X_4, X_5, X_3$	$X_{12}: X_{10}, X_9, X_{11}, X_7$	$X_{18}: X_{16}, X_{17}, X_{11}, X_7, X_{15}$

我们可以证明，该算法在进入第二个周期之后，必定会在某个n值处停下来，也就是说，令算法终止的那个n其值小于$\mu+2\lambda$。更多详情可参考［Knu2］。

标尺函数揭示了汉诺塔问题[⊖]的解法。可将n个圆盘由小至大分别标上序号0至$n-$

⊖ Tower of Hanoi。从左至右有A、B、C三根柱子，一开始A柱从上至下叠放了n个中间带孔的圆盘，每个圆盘的面积均比上面一个大。现在要花2^n-1步将这些盘子移至C柱。每次只准移动一个圆盘，而且不得将大圆盘叠放在小圆盘之上。详情参见：http：//zh.wikipedia.org/wiki/汉诺塔。——译者注

1，在第 k 步移动时（k 的取值从 1 至 2^n-1），我们把编号为 ntz(k) 的那个圆盘循环右移至距其最近的柱子上[⊖]。

标尺函数还可以用来生成反转的二进制格雷码（参见 13.1 节）：对于任意长度为 n 的字组，每次执行第 k 步时（k 的取值从 1 至 2^n-1），都将标号为 ntz(k) 的那个位元值反转。

5.5 习题

1. 用 Dubé 算法求 ntz 函数，并将其中的乘法操作展开为加法及移位操作。
2. 用 3 条基本 RISC 指令实现“右对齐”函数（right justify function）：$\boldsymbol{x} \overset{u}{\gg} \text{ntz}(\boldsymbol{x})$，$\boldsymbol{x} \neq \boldsymbol{0}$。
3. 并行前缀操作与并行后缀操作（以异或操作为例）是否可逆？如果可以的话，如何求出反函数？

⊖ 意思就是，假如第 k 步待移动的圆盘 ntz(k) 位于 B 柱，那么若其右侧 C 柱没有圆盘，或是最上方圆盘的面积大于该盘，则将其移到 C 柱，否则就回过头来移至 A 柱。编号为 ntz(k) 的圆盘位于 A、C 两柱时，以此类推。——译者注

第 6 章

在字组中搜索位串

6.1 寻找首个值为 0 的字节

这个函数的出现主要源于 C 语言表示字符串的方式。在 C 语言中，字符串的长度并不直接随字符串存放，而是在字符串最后附加一个“全 0 字节”[⊖]。C 语言中，我们用“strlen”（string length，“字符串长度”一词的简写）函数获取字符串长度。此函数从左至右搜寻字符串，直至发现 0 值字节，然后返回除 0 值字节外已扫描的字节数。

有一种快速实现“strlen”函数的方法是：每次载入并测试单一字节，直至达到字组边界，然后把这些字节一次性装入寄存器中，并测试寄存器中是否存在 0 值字节。在大端序的计算机中，我们要从左至右求出首个 0 值字节的下标。为了方便求值，可以用 0 至 3 分别代表下标为 0 到 3 的字节是 0 值字节，用 4 表示字组中没有 0 值字节。这样就可以先把字符串长度初始化为 0，在后续搜索过程中，持续为字符串长度加上一个表示当前字组中 0 值字节位置的编码。因为小端序的计算机按照与大端序计算机相反的方式把字组载入寄存器，所以我们要从右至左求出首个 0 值字节在寄存器内的下标。这里我们专注于下面两个函数，其中“00”表示 0 值字节，“nn”表示非 0 值的字节，“xx”表示其值可能为 0 也可能非 0 的字节。

$$\text{abytel}(x)=\begin{cases}0, & x=\text{00xxxxxx},\\ 1, & x=\text{nn00xxxx},\\ 2, & x=\text{nnnn00xx},\\ 3, & x=\text{nnnnnn00},\\ 4, & x=\text{nnnnnnnn}.\end{cases}\qquad \text{zbyter}(x)=\begin{cases}0, & x=\text{xxxxxx00},\\ 1, & x=\text{xxxx00nn},\\ 2, & x=\text{xx00nnnn},\\ 3, & x=\text{00nnnnnn},\\ 4, & x=\text{nnnnnnnn}.\end{cases}$$

图 6.1 列出了第 1 种求最左侧 0 值字节的方法。该算法只是从左至右简单地测试每个字节，直至找到首个 0 值字节并返回结果。

⊖ all-0 byte，也就是每个位元均为“0”的字节。——译者注

```
int zbytel(unsigned x) {
   if              ((x >> 24) == 0) return 0;
   else if ((x & 0x00FF0000) == 0) return 1;
   else if ((x & 0x0000FF00) == 0) return 2;
   else if ((x & 0x000000FF) == 0) return 3;
   else return 4;
}
```

图 6.1 用简单的测试序列来寻找最左侧的 0 值字节

该算法需要执行 2 至 11 条基本 RISC 指令，如果字组中没有 0 值字节（这种情况对 strlen 函数很重要），那么就得执行 11 条指令。寻找最右侧 0 值字节的代码与之非常相似。

图 6.2 用无分支代码实现了此函数。该算法的思路是：将每个 0 值字节转化成 0x80，将每个非 0 字节转换为 0x00，然后使用前导 0 计数函数。如果计算机支持前导 0 计数指令及“或非”指令，则此算法要执行 8 条指令。[Lamp] 描述了一些与之类似的技巧。

```
int zbytel(unsigned x) {
   unsigned y;
   int n;
                          // Original byte: 00 80 other
   y = (x & 0x7F7F7F7F) + 0x7F7F7F7F;   // 7F 7F 1xxxxxxx
   y = ~(y | x | 0x7F7F7F7F);           // 80 00 00000000
   n = nlz(y) >> 3;             // n = 0 ... 4, 4 if x
   return n;                    // has no 0-byte.
}
```

图 6.2 用无分支代码寻找最左侧的 0 值字节

在上述算法求出的 y 值中，将后缀 0 的个数除以 8，并舍去小数部分，就能得出 x 最右侧的 0 值字节位置。通过前导 0 计数函数能够求出后缀 0 的个数（参阅 5.4 节），利用此表达式，我们可以把算法中对 n 赋值的那一行替换为：

```
n = (32 - nlz(~y & (y - 1))) >> 3;
```

如果计算机有“nor”（或非）及“and not”指令，那么此方案仅需 12 条指令。

顺便说一下，在大部分 PowerPC 架构的计算机中，都用不到寻找最右侧 0 值字节的程序，因为可以通过“反字节序字组加载”指令（load word byte-reverse instruction，lwbrx）来加载字组，然后求其最左侧 0 值字节的位置即可。

图 6.2 在 64 位计算机上的价值要比在 32 位机上大，因为经过修改后，适用于 64 位机的程序所需指令数与 32 位版本相同（7 至 10 条，具体数值取决于常数加载所需的指令数），而图 6.1 所用算法却需要 23 条指令。

如果只想判断是否存在 0 值字节，那么可以在执行完第 2 条对 y 的赋值语句后，插入一条“零分支”（branch on zero）或“非零分支”（branch on nonzero）指令。

[Mycro] 给出了一个与图 6.2 相似的算法，只不过这次是要寻找 x 最右侧的 0 值字节（也就是求 zbyter(x) 的值）：

```
y = (x - 0x01010101) & ~x & 0x80808080;
n = ntz(y) >> 3;
```

如果计算机支持“and not”指令与后缀 0 计数指令，那么除去加载常数的开销，这个算法只需 5 条指令。由于存在借位问题，所以它不能用来计算 zbytel(x) 的值。此算法最用有之处就是可以在小端序的计算机中寻找字符串里的 0 值字节，或是在任意端序的计算机上判断是否存在 0 值字节（只用向 y 赋值的那样语句即可）。

若计算机不支持前导 0 计数指令，那么恐怕没有“寻找首个 0 值字节”（find first 0-byte function）的好办法。图 6.3 列出了一种可以使用的算法（只列出了可执行代码部分）。

该算法需要 10 至 13 条基本 RISC 指令。如果碰到全部字节都不是 0 的情况，那么需要 10 条。尽管它的分支数比图 6.1 少，但是它恐怕不如那个算法好。而且不巧的是，这段算法并不适用于 64 位计算机。

还有其他几种方法能避免使用 nlz 函数。图 6.3 求出的 y 值共有 4 个字节，每个字节的值不是 0x00 就是 0x80。y 中最多有 4 个值为“1”的位元，而将该数除以 15 后，这些位元就会全部压缩至余数最右侧的 4 个位置上。因此，在模除的结果中，0 至 15 这 16 个数与原数的 16 种取值存在一一对应关系。例如：

remu(**0x8080 8080**,**127**)=**15**,

remu(**0x8000 0000**,**127**)=**8**,

remu(**0x0000 8080**,**127**)=**3**，等等。

将该值作为索引，来查一张长度为 16 的表格，即可求出结果。因此，以 if(y==0) 开始的数行代码可以替换为：

```
static char table[16] = {4, 3, 2, 2, 1, 1, 1, 1,
                         0, 0, 0, 0, 0, 0, 0, 0};
return table[y%127];
```

其中 y 是无符号数。也可以用 31 代替 127，不过那样做需要另外一套表格数据。

```
                     // Original byte: 00 80 other
y = (x & 0x7F7F7F7F) + 0x7F7F7F7F; // 7F 7F 1xxxxxxx
y = ~(y | x | 0x7F7F7F7F);         // 80 00 00000000
                                   // These steps map:
if (y == 0) return 4;              // 00000000 ==> 4,
else if (y > 0x0000FFFF)           // 80xxxxxx ==> 0,
   return (y >> 31) ^ 1;           // 0080xxxx ==> 1,
else                               // 000080xx ==> 2,
   return (y >> 15) ^ 3;           // 00000080 ==> 3.
```

图 6.3 在不使用 nlz 指令的前提下寻找最左侧的 0 值字节

涉及和 127 或 31 模除的那两种算法只不过是好玩罢了。因为求余函数即使以硬件方式实现，也至少需要执行 20 个周期，然而，如果把图 6.3 中以 if (y==0) 开始的语句替换为下列两种形式，那么算法的效率就会高很多：

```
return table[hopu(y, 0x02040810) & 15];
return table[y*0x00204081 >> 28];
```

此处的 hopu(a, b) 指的是无符号数 a 与 b 乘积的高 32 位。在第 2 行代码中，我们假定乘法操作遵守通常的“HLL 约定”，也就是说，乘法操作的结果就是完整乘积的低 32

位。这种方法比较实际一些，因为在支持快速乘法的计算机上，我们可以直接运算，在不支持时，也能用移位及加法操作模拟乘法。用 4 条指令即可实现乘法运算，因为它可以分解为下列形式：

$$y(1+2^7+2^{14}+2^{21})=y(1+2^7)(1+2^{14})$$

采用上述方式计算乘法只需 4 个周期，这样整个算法用 13 个周期就可执行完（计算 y 用 7 个周期，移位及加法操作用 4 个周期，右移 28 位并以此为下标查表用 2 个周期），而且代码还是无分支的。

这几种算法都可以扩展至 64 位计算机。64 位机的“模除式算法”可使用下列语句：

```
return table[y%511];
```

其中 table 长度为 256，其中各项的值为 8、0、1、0、2、0、1、0、3、0、1、0、2、0、1、0、4、…（也就是说，table［i］的值就是 i 中后缀 0 的个数）。

对于那两种使用乘法操作的算法，可以改写为：

```
return table[hopu(y, 0x02040810 20408100) & 255]; 或者
return table[(y*0x00020408 10204081)>>56];
```

其中 table 长度是 256，其中的值是 8、7、6、6、5、5、5、5、4、4、4、4、4、4、4、4、3、…[1]。

与 0x20408 10204081 的乘法可以用系列方法实现：

$$t_1 \leftarrow y(1+2^7)$$
$$t_2 \leftarrow t_1(1+2^{14})$$
$$t_3 \leftarrow t_2(1+2^{28})$$

这样用 13 个周期就能执行完了。

在所有使用查表法的变种算法里，只要把表格数据简单替换一下，就可以用来实现“寻找最右侧 0 值字节”函数了。

若计算机没有“或非”指令，那么图 6.3 中，对 y 赋值的那行语句便可以省去“非”操作。在 32 位计算机中，采用上述三种 return 语句之一即可求出函数值。表格元素 table[i] 的值为 0、0、0、0、0、0、0、0、1、1、1、1、2、2、3、4。这种方法不适用于 64 位机。

图 6.2 的算法还有一种有趣的变种，也适用于不支持前导 0 计数指令的计算机。设 a、b、c、d 均为 1 位元变量，它们分别表示“$\boldsymbol{x}$ 首字节非 0”、“$\boldsymbol{x}$ 次字节非 0”等谓词的真假。则由此可得：

$$\text{zbytel}(\boldsymbol{x})=\boldsymbol{a}+\boldsymbol{ab}+\boldsymbol{abc}+\boldsymbol{abcd}$$

乘法操作可用“与”运算实现，这样就得出了图 6.4 中的算法（只列出可执行代码）。此算法需要 15 条基本 RISC 指令，虽说不是特别快，但却具备一定量的并发执行度。如

[1] table［i］的值是 $8-\lceil \log_2(i+1)\rceil$。——译者注

果一台超标量计算机可同时执行 3 条相互独立的算数指令，那么此算法只需 10 个周期就能执行完。

```
y = (x & 0x7F7F7F7F) + 0x7F7F7F7F;
y = y | x;            // Leading 1 on nonzero bytes.

t1 =  y >> 31;                // t1 = a.
t2 = (y >> 23) & t1;          // t2 = ab.
t3 = (y >> 15) & t2;          // t3 = abc.
t4 = (y >>  7) & t3;          // t4 = abcd.
return t1 + t2 + t3 + t4;
```

图 6.4　通过求多项式的值来查找最左侧的 0 值字节

将该算法简单变换一下，就可据此求出“最右侧 0 值字节”函数了：

$$\text{zbyter}(x) = \boldsymbol{abcd} + \boldsymbol{bcd} + \boldsymbol{cd} + \boldsymbol{d}$$

（上述算法比图 6.4 多一次“与”操作。）

6.1.1　0 值字节位置函数的一些简单推广

zbytel 与 zbyter 函数可以用来查找与特定值相符的字节：首先构建一个字组，把待查找的值复制到其中的每个字节上，然后将参数 x 与该字组取异或。例如，要搜索 x 中是否有 ASCII 码的空格字符（值为 0x20），那么只需搜索 $x \oplus$**0x2020 2020** 中是否有 0 值字节即可。

与此类似，若想在 x 与 y 中查找值相同的字节，只需在 $x \oplus y$ 的结果中搜索 0 值字节就好。

图 6.2 中的代码及其变形算法对字节边界没有特殊要求。举例来说，如果要在字组 x 中搜索前 4 个位元、接下来的 12 个位元以及倒数 16 个位元中有没有“0”，那么只需把图 6.2 中的掩码替换为 0x77FF 7FFF 即可［PHO］。（将待搜索位段的首个位元所对应的掩码位置 0。）

6.1.2　搜索给定范围内的值

简单修改一下图 6.2，就可以用它来判断是否存在值位于 0 至小于 128 的特定值之间的字节。为了演示此算法，我们用下列代码找出值位于 0 至 9 之间的最左侧字节下标：

```
y = (x & 0x7F7F7F7F) + 0x76767676;
y = y | x;
y = y | 0x7F7F7F7F;           // Bytes > 9 are 0xFF.
y = ~y;                       // Bytes > 9 are 0x00,
                              // bytes <= 9 are 0x80.
n = nlz(y) >> 3;
```

在更通用的情况下，我们想找出字组中值位于 a、b 之间的最左侧字节，此处 a 与 b 的间距小于 128。举例来说，ASCII 码中的大写字母都位于 0x41 至 0x5A 之间。想要找出字组中首个大写字母，我们可以把它和 0x4141 4141 相减，并设法使可能产生的借位不会越过字节边界。然后再用上述代码找出值位于 0 至 0x19（0x5A－0x41）之间的字节。用 2.18 节给出的多字节减法公式，再结合 $\boldsymbol{y}$=0x4141 4141 进行化简[⊖]，可得如下代码：

⊖ 公式中本来要对 $\boldsymbol{y}$ 和 0x7F7F 7F7F 做“与”操作，以屏蔽每个字节的最高位，但由于 $\boldsymbol{y}$ 的值 0x4141 4141 的每个字节最高位本来就是 0，所以不需要再屏蔽一次了。——译者注

```
d = (x | 0x80808080) - 0x41414141;
d = ~((x | 0x7F7F7F7F) ^ d);
y = (d & 0x7F7F7F7F) + 0x66666666;
y = y | d;
y = y | 0x7F7F7F7F;     // Bytes not from 41-5A are FF.
y = ~y;                 // Bytes not from 41-5A are 00,
                        // bytes from 41-5A are 80.
n = nlz(y) >> 3;
```

对于某些范围来说，此代码还可继续简化。例如想找出首个值在 0x30 至 0x39 之间（十进制中“0”至“9”这 10 个数字的 ASCII 编码）的字节，只需要把输入的字组同 0x3030 3030 取异或，然后再用前面讲过的代码判断是否有值在 0 和 9 之间的字节即可。（进行此种简化的前提是：上下限的 n 个高权重位元相同，且下限值的最后 $8-n$ 个位元都是“0”。）

不需要再添加指令，就可以将这些算法应用在搜索范围大于等于 128 的情况下。举例来说，如果要找出最左方且值在 0 至 137(0x89) 之间的字节下标，只需把适用于 0 至 9 的算法中 y=y|x 这行代码改成 y=y&x 就可以了。

与之类似，如果要查找值位于 0x41 至 0xDA 之间的最左侧字节，那么只需把适用于 0x41 至 0x5A 的代码中 y=y|d 这一行改为 y=y&d 即可。

6.2 寻找首个给定长度的全 1 位串

这里的问题是，要在寄存器中的字组里搜索一个长度至少为 n，且每个位元均为“1”的位串，并返回其位置，如果找不到，则返回特殊值。这个问题有两个变种，一个是只返回“是/非”值来表示查找结果，另一个是只搜寻长度恰好为 n 的“全 1 位串”(string of 1’s)。该问题会用在磁盘分配程序之中，尤其是进行磁盘压缩（重新整理磁盘中的数据，连续排列每个文件的所有存储块）时。Albert Chang 建议我谈谈这个问题，他认为这是前导 0 计数指令的一项应用。

我们假定计算机要么支持前导 0 计数指令，要么有一套合适的子程序能够实现此函数。

大家一开始肯定能想到一种算法：先计算前导 0 的个数，然后通过左移操作跳过这些位元，最后将该值取反并计算前导 0 的个数，以此求出前导 1 的个数。如果此数量符合要求，则结束算法。否则就把该值按计算出来的数量左移位，并返回算法开头，继续查找。此算法的代码如下。若是找到了 n 个连续出现且值为“1”的位元，则返回一个 0 至 31 之间的数，用来表示从左方算起首个符合条件的位串中第一个值为“1”的位元所处的位置。否则就返回 32 以表示找不到这样的位串。

```
int ffstr1(unsigned x, int n) {
   int k, p;

   p = 0;                  // Initialize position to return.
   while (x != 0) {
```

```
        k = nlz(x);          // Skip over initial 0's
        x = x << k;          // (if any).
        p = p + k;
        k = nlz(~x);         // Count first/next group of 1's.
        if (k >= n)          // If enough,
           return p;         // return.
        x = x << k;          // Not enough 1's, skip over
        p = p + k;           // them.
     }
     return 32;
  }
```

若循环的执行次数不多，则该算法较为合适。比方说，假如我们知道x中会出现长串的“1”及长串的“0”，那么就可以使用该算法。磁盘分配程序很可能就属于刚说的这种情况。然而它在最坏情况下的执行时间则不是很好。例如，当x＝0x5555 5555且$n \geqslant 2$时，算法就需要执行大约178条完整的RISC指令集中的指令。

另外一种算法在最坏情况下的执行时间比上述算法好，它基于左移位及加法指令。为了演示此算法，我们考虑在长度为32的字组x中，寻找长度至少为8的“全1位串”。此时的计算过程如下：

$$x \leftarrow x \ \&(x \ll 1)$$
$$x \leftarrow x \ \&(x \ll 2)$$
$$x \leftarrow x \ \&(x \ll 4)$$

在执行完第1次赋值之后，x中值为“1”的位元表示有长度为2的“全1位串”始于此处。在执行完第2次赋值之后，x中值为“1”的位元则表示有长度为4的位串始于此位置（也就是一个长度为2的位串后面跟着另一个长度为2的位串）。执行完第3次赋值之后，x中值为“1”的位元则表示有长度为8的位串始于此处。此时求出x中前导0的个数，也就可以知道首个长度大于等于8的位串从何处开始了。若前导0的个数是32，则表示没有这种位串。

为了研究出一套适用于任意长度n（n在1至32之间）的算法，我们以一种稍微不同的方式来重新观察上述过程。首先可以肯定，上面3次赋值操作的顺序可以打乱。于是我们将其进行逆序排列，这样看上去就更方便了。为了演示通用算法，此处考虑$n=10$的情况：

$$x_1 \leftarrow x \ \ \&(x \ll 5)$$
$$x_2 \leftarrow x_1 \ \ \&(x_1 \ll 2)$$
$$x_3 \leftarrow x_2 \ \ \&(x_2 \ll 1)$$
$$x_4 \leftarrow x_3 \ \ \&(x_3 \ll 1)$$

第1条语句是将x左移$n/2$位。执行完之后，问题就简化成在x_1中寻找连续出现5个“1”的位串。要求解这个问题，可以把x_1左移$\lfloor 5/2 \rfloor = 2$位，然后再和x_1本身求“与”，并在运算结果中寻找长度为3（5－2）的位串。后两条语句就可以找出x_2中是否存在长度为3的位串。所有语句的移位量总和永远是$n-1$。该算法的代码列在图6.5之中。在n位于1至32之间时，该算法需要执行3至36条完整的RISC指令。

如果 n 的值比较大，那么也可以考虑把循环展开，将原有的循环体重复执行 5 次，并省略对 n>1 所作的判断。（在 32 位计算机上，执行 5 次足够了。）这种无分支算法的执行时间是恒定的，永远都是 20 条指令（对 n 的最后一次赋值可以忽略）。若 n 值较小，则那 3 条赋值语句的执行次数会比实际需要的次数多，虽说如此，但是这些额外步骤并不改变运算结果。因为执行到一定程度之后，n 的值就一直都是 1 了[㊀]，这样的话，那三条赋值语句对 x 或 n 的值就都没有影响了。如果以执行的指令数来衡量，那么在 n≥5 时，展开版本比循环版本快。

```
int ffstr1(unsigned x, int n) {
   int s;

   while (n > 1) {
      s = n >> 1;
      x = x & (x << s);
      n = n - s;
   }
   return nlz(x);
}
```

图 6.5 通过一系列移位及加法操作，寻找首次出现的“全 1 位串”

如果要判断 $\boldsymbol{x}$ 中是否有长度恰好为 n 的“全 1 位串”，那么还要多用 6 条指令（如果支持“与非”指令，那就是多用 4 条）。在图 6.5 计算出来的 $\boldsymbol{x}$ 中，值为“1”的位元表示有长度至少为 n 的“全 1 位串”始于此处。因此，可将上述算法最终求出的 $\boldsymbol{x}$ 值代入下面这个表达式：

$$\boldsymbol{x} \,\&\, \neg(\boldsymbol{x} \overset{u}{\gg} 1) \,\&\, \neg(\boldsymbol{x} \ll 1)$$

如果现在的 $\boldsymbol{x}$ 中出现值为“1”的孤立位元[㊁]，则说明在起初的 $\boldsymbol{x}$ 中，有长度恰好为 n 的“全 1 位串”始于此处。

经过简单修改，此算法也能用于在特定位置上查找长度为 n 的“全 1 位串”。举例来说，要寻找始于字节边界处的位串，可以将最终求出的 $\boldsymbol{x}$ 值同 0x8080 8080 做“与”运算。

此算法也可用于寻找由“0”所组成的位串：可以在开始执行算法时先把 $\boldsymbol{x}$ 取反，或是将算法中的“与”操作改为“或”操作，并在调用 nlz 函数之前对其取反。比如说，下面这个算法就可以找出首个（也就是最左侧的）0 值字节（此问题的准确定义请参看 6.1 节）。

$\boldsymbol{x} \leftarrow \boldsymbol{x} \mid (\boldsymbol{x} \ll 4)$

$\boldsymbol{x} \leftarrow \boldsymbol{x} \mid (\boldsymbol{x} \ll 2)$

$\boldsymbol{x} \leftarrow \boldsymbol{x} \mid (\boldsymbol{x} \ll 1)$

$\boldsymbol{x} \leftarrow \mathbf{0x7F7F7F7F} \mid \boldsymbol{x}$

$\boldsymbol{p} \leftarrow \mathrm{nlz}(\neg\boldsymbol{x}) \overset{u}{\gg} 3$

上述算法要执行 12 条完整 RISC 指令（这个算法不如图 6.2 所述的算法好，后者只需执行 8 条指令）。

㊀ 发生此状况的原因是，当 n 第一次降为 1 时，s = n≫1 算出的 s 值会是 0，于是不论再执行多少次 n = n − s 及 s = n≫1，s 的值一直是 0，而 n 的值一直是 1。——译者注

㊁ isolated 1−bit。也就是说，如果该位元在字组最左侧，则其右侧位元是“0”，如果该位元在字组最右侧，则其左侧位元是“1”，其余情况下，其左右两侧位元皆为“0”。——译者注

6.3 寻找最长全1位串

图6.6所列函数非常简洁，它可以找到x中最长“全1位串”的长度[Hsieh]。它需要执行4n+3条基本RISC指令，其中n代表最长“全1位串”的长度。在最坏情况下需要131条指令。

```
int maxstr1(unsigned x) {
   int k;
   for (k = 0; x != 0; k++) x = x & 2*x;
   return k;
}
```

图6.6 寻找最长“全1位串”的长度

为减少最坏情况下的执行时间，可以采用该算法的“对数版”(logarithmic version)。它将原数中值是“0”的位元向左传播1、2、4、8、16位，直到整个字组变为0，然后回溯并找出最长的连续“全1位串”之长度。

比方说，假设

```
x = 0011 1111 1111 0011 1111 0011 1111 1000
```

那么

```
 x2 = 0011 1111 1110 0011 1110 0011 1111 0000
 x4 = 0011 1111 1000 0011 1000 0011 1100 0000
 x8 = 0011 1000 0000 0000 0000 0000 0000 0000
x16 = 每个单位都是“0”
```

在本例中，最后一个非零值是x8。其中每个值为“1”的位元都表示原数中长度为8的“全1位串”左界。因此，原数中最长的“全1位串”就始于x8中最左边的那个值为“1”的位元处，在本例中，也就是第29位。先检测x4的第21位(29−8)，以判断原数中有没有长度为12的“全1位串”。因为该位是“0”，所以原数中没有长度是12的“全1位串”。然后检测x2的21号位，判断原数中是否有长度是10的“全1位串”。由于该位是“1”，所以在29号位有一个长度至少为10的“全1位串”。最后检测x的19号位(21−2)，判断这个位串长度是否为11。由于该位是“0”，所以原数中最长的“全1位串”其长度为10，始于29号位。

图6.7列出了上述算法的代码，不过它只用了x和y两个变量，而非x、x2、x4、x8、x16等5个变量。这段代码既能找到最长“全1位串”的长度，也能找到其左端点的位置。若x是0，或所有位元都是“1”，则该算法无效。这两种情况都加以特别处理了，后者放在了一个不常执行的分支中。

```
int fmaxstr1(unsigned x, int *apos) {
   unsigned  y;
   int s;

   if (x == 0) {*apos = 32; return 0;}
   y = x & (x << 1);
   if (y == 0) {s = 1; goto L1;}
   x = y & (y << 2);
   if (x == 0) {s = 2; x = y; goto L2;}
   y = x & (x << 4);
   if (y == 0) {s = 4; goto L4;}
   x = y & (y << 8);
   if (x == 0) {s = 8; x = y; goto L8;}
   if (x == 0xFFFF8000) {*apos = 0; return 32;}
   s = 16;

L16: y = x & (x << 8);
     if (y != 0) {s = s + 8; x = y;}
L8:  y = x & (x << 4);
     if (y != 0) {s = s + 4; x = y;}
L4:  y = x & (x << 2);
     if (y != 0) {s = s + 2; x = y;}
L2:  y = x & (x << 1);
     if (y != 0) {s = s + 1; x = y;}
L1:  *apos = nlz(x);
   return s;
}
```

图6.7 寻找最长“全1位串”的长度及起始位置

该算法最坏情况下需执行39条基本RISC指令，此外还要算上实现nlz函数所

用的那些。就算只求出最长“全 1 位串”的长度而不寻找起始位置也节省不了太多时间，仅可免于实现 nlz 函数而已。

6.4 寻找最短全 1 位串

在字组中寻找最短“全 1 位串”（the shortest string of 1-bits）要更难些。一种方法是把字组中所有“全 1 位串”的起始位置标在字组 b 中，结束位置标在字组 e 中。然后，若 b & e 非零，则最短“全 1 位串”长度就是 1。否则，将 e 左移一位，继续测试。例如，假设

```
x = 0011 1111 1111 0011 1111 0011 1111 1000
```

那么

```
b = 0010 0000 0000 0010 0000 0010 0000 0000
e = 0000 0000 0001 0000 0001 0000 0000 1000
```

把 e 左移 5 次后，b & e 非零，这说明原字组中最短“全 1 位串”长度是 6。

图 6.8 中的代码体现了计算过程。如上所述，位串位置从原字组左方算起，若有至少两个长度相等的最短“全 1 位串”，则此函数返回最左边那个。比方说，如果 x = 0x00FF 0FF0，则返回的长度是 8，位置也是 8。

```
int fminstr1(unsigned x, int *apos) {
   int k;
   unsigned b, e;        // Beginnings, ends.

   if (x == 0) {*apos = 32; return 0;}
   b = ~(x >> 1) & x;    // 0-1 transitions.
   e = x & ~(x << 1);    // 1-0 transitions.
   for (k = 1; (b & e) == 0; k++)
      e = e << 1;
   *apos = nlz(b & e);
   return k;
}
```

图 6.8 寻找最短“全 1 位串”的长度与起始位置

在 $n \geqslant 2$ 时此函数需执行 $8+4n$ 条基本 RISC 指令（基本 RISC 指令集中无 *andc* 指令），另外还要算上 nlz 函数的执行时间，其中 n 是 x 中最短“全 1 位串”的长度。

这类问题也许最终都可以归结为：给定一个大于 0 的整数 n，找出一个长度不小于它的最短“全 1 位串” $\boldsymbol{x}$，返回其长度与位置。从“存储分配问题”（storage allocation problem）的角度看，这是种“最佳适配”算法（“best fit” algorithm）。其解法是，先将 $\boldsymbol{x}$ 中的“0”向左传播 $n-1$ 位，然后在修改过的 $\boldsymbol{x}$ 中寻找最短“全 1 位串”。详情参见习题。

6.5 习题

1. 加工 Hsieh 算法，使其同时找出 $\boldsymbol{x}$ 中最长“全 1 位串”的长度及起始位置。可以使用 nlz 函数。

2. 编写一个函数，在字组 $\boldsymbol{x}$ 中找到长度不小于给定整数 n 的最短“全 1 位串”，输出其长度及位置。
3. 还有一种寻找 $\boldsymbol{x}$ 中最短“全 1 位串”的方法，就是将最右侧的“全 1 位串”置 0，观察该操作前后种群计数的变化量，并反复执行此步骤。用上述思路编写一个使用完整 RISC 指令集的函数，找出最短“全 1 位串”的长度及位置。
4. 对于“完全随机”（completely random，也就是每个位元值为“0”及“1”的概率各半，且相互独立）的 32 位字组 $\boldsymbol{x}$ 来说，其“全 1 位串”的平均个数是多少？如果用这样的 $\boldsymbol{x}$ 作为习题 3 的输入数据，那么此均值就决定了那个算法的平均执行时间。
5. 对于“完全随机”的 32 位字组 $\boldsymbol{x}$ 来说，其最短“全 1 位串”的平均长度是多少？如果用这样的 $\boldsymbol{x}$ 作为图 6.8 中 `fminstr1` 函数的输入数据，那么此均值就决定了那个函数的平均执行时间。用“蒙特卡罗算法”（Monte Carlo）[㊀] 或穷举排列算法（exhaustive enumeration）编写程序求出该值。
6. 在长度为 n 位元的 2^n 个二进制数中，长度为 1 的最短“全 1 位串”有多少个？即有多少个以“10”开头，或以“01”结尾，或包含序列“010”的 n 位元二进制数？勿使用穷举法编程，找到此问题的“闭合式解法”（closed-form solution）[㊁] 或递归解法。
7. 与上题相似，在长度为 n 位元的 2^n 个二进制数中，长度是 2 的最短“全 1 位串”有多少个？

㊀ 也称统计模拟方法，是 1946 年左右由约翰·冯·诺依曼（John von Neumann，1903—1957）、斯塔尼斯拉夫·乌拉姆（Stanislaw Ulam，1909—1984）、尼古拉·梅特罗波利斯（Nicholas Metropolis，1915—1999）等人在“洛斯阿拉莫斯国家实验室”为“曼哈顿核武计划”工作期间提出的数值计算方法，以概率统计理论做指导。传说乌拉姆的叔叔常在摩纳哥蒙特卡罗赌场借钱赌博，而该算法正以概率论为基础，故得此名。它使用随机数或更常见的伪随机数来解决很多计算问题，广泛运用于金融工程学、宏观经济学、生物医学、计算物理学（如粒子输运计算、量子热力学计算、空气动力学计算）等领域。在数学中可用其估算积分和圆周率。详情参见：https://zh.wikipedia.org/wiki/蒙特卡罗方法。——译者注

㊁ 也叫“闭式解”、“解析解”，是指经有限次“常见运算”（well-known method）而得的解法。详情参见：https://en.wikipedia.org/wiki/Closed-form_solution。——译者注

第 7 章

重排位元与字节

7.1 反转位元与字节

“反转位元”（reverse bit）的意思是将寄存器中的内容依中点对换位置[⊖]，例如：

$$\text{rev}(\mathbf{0x01234567}) = \mathbf{0xE6A2C480}$$

“反转字节”（reverse byte）的意思与之类似，就是反向排列寄存器中的 4 个字节。要想把 DEC 及 Intel 处理器使用的“小端序”（little-endian）数据转换成其他 CPU 使用的“大端序”（big-endian），就必须学会字节反转。

以两个位元为一组，对调其中的相邻位元；再以 4 个位元为一组，将之编为两个位段，每个位段含有两个位元，并对调相邻位段，以此类推，即可高效反转字节。下面列出了该算法［Aus1］代码。这 5 条赋值语句可按任意顺序执行。这与 5.1 节所列的第一个种群计数算法相同，只是其中的加法操作现在改为对换位置操作。

```
x = (x & 0x55555555) <<  1 | (x & 0xAAAAAAAA) >>  1;
x = (x & 0x33333333) <<  2 | (x & 0xCCCCCCCC) >>  2;
x = (x & 0x0F0F0F0F) <<  4 | (x & 0xF0F0F0F0) >>  4;
x = (x & 0x00FF00FF) <<  8 | (x & 0xFF00FF00) >>  8;
x = (x & 0x0000FFFF) << 16 | (x & 0xFFFF0000) >> 16;
```

可以稍微改善上述算法，不使用那么多大常数，并把最后两条赋值语句写得更直观些，这样就有了图 7.1 所示的算法（需 30 条无分支的基本 RISC 指令）。

代码中最后一个 x 赋值语句需要 9 条基本 RISC 指令。若 CPU 支持循环移位，则只需 7 条。

$$x \leftarrow ((x \,\&\, \mathbf{0x00FF00FF}) \overset{rot}{\gg} 8) \mid (x \overset{rot}{\gg} 8) \,\&\, \mathbf{0x00FF00FF})$$

⊖ 范例中 0x0123 4567 的二进制是 0000 0001 0010 0011 0100 0101 0110 0111，依中点对换位置，也就是将最左方的 31 号位元与最右方的 0 号位元对调，将次左方的 30 号位元与次右方的 1 号位元对调，以此类推，直至 16、15 两位置。对调之后的二进制数是 1110 0110 1010 0010 1100 0100 1000 0000（这个数实际上也就是将前面那个二进制数逆序写出），十六进制是 0xE6A2 C480。——译者注

```
unsigned rev(unsigned x) {
   x = (x & 0x55555555) <<  1 | (x >>  1) & 0x55555555;
   x = (x & 0x33333333) <<  2 | (x >>  2) & 0x33333333;
   x = (x & 0x0F0F0F0F) <<  4 | (x >>  4) & 0x0F0F0F0F;
   x = (x << 24) | ((x & 0xFF00) << 8) |
       ((x >> 8) & 0xFF00) | (x >> 24);
   return x;
}
```

图 7.1 反转位元

PowerPC 只用 3 条指令就可反转字组中的 4 个字节［Hay1］：先循环左移 8 位，这样有两个字节已经就位了，然后再用两条“rlwimi”指令（rotate left word immediate then mask insert，用立即数做移位量，根据掩码将字组循环左移）[⊖]即可。

下面再介绍一个由 Christopher Strachey 先生给出的算法［Strach 1961］。从计算机算法的角度看，它有些过时了，但仍能启发思路。它假定左侧 16 个位元在算法执行之初业已清零，算法完成时，反转后的半字组出现在寄存器的左半区中。

该算法基于每个位元必须移动的长度。为了算出结果，从左至右的这 16 个位元必须分别移动 1，3，5，…，31 个位置才行。我们先移动那些移位量大于等于 16 的位元，然后移动那些移位量大于等于 8 的位元，以此类推。该算法演示如下，其中每个字母代表一个二进制位，句点表示此位置上的位元与该算法无关。

```
0000 0000 0000 0000 abcd efgh ijkl mnop  Given
0000 0000 ijkl mnop abcd efgh .... ....  After shl 16
0000 mnop ijkl efgh abcd .... .... ....  After shl 8
00op mnkl ijgh efcd ab.. .... .... ....  After shl 4
0pon mlkj ihgf edcb a... .... .... ....  After shl 2
ponm lkji hgfe dcba .... .... .... ....  After shl 1
```

直观的实现代码需要 16 条基本 RISC 指令，再加 12 条常数加载指令：

```
x = x | ((x & 0x000000FF) << 16);
x = (x & 0xF0F0F0F0) | ((x & 0x0F0F0F0F) << 8);
x = (x & 0xCCCCCCCC) | ((x & 0x33333333) << 4);
x = (x & 0xAAAAAAAA) | ((x & 0x55555555) << 2);
x = x << 1;
```

用原有常数的补码做掩码可以减少掩码种数。若采用非常规的掩码，则可保留寄存器右侧 16 个位元。

若能使用循环移位，则可根据 Strachey 算法的思路来反转 32 位字组。我们考虑每个位元必须循环左移多少位才能到达其最终位置。从左到右看，这 32 个位元的移位量分别是 1，3，5，…，31，1，3，5，…，31（所有位元的移位量都不可能是偶数）。这个算法先移动那些移位量大于等于 16 的位元，然后再移那些移位量大于等于 8 的位元，以此类推，最后移动那些移位量是 1 的位元（实际上所有位元在这一步都要移动，因为它们前面所经历的移位总量是偶数，所以必须再移一次，才能变成奇数）。反转 32 位字组 x 所用的

⊖ 详情参见：http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.aixassem/doc/alangref/rlwimi.htm。——译者注

算法如下。函数 shlr（x,y）表示将 x 循环左移 y 位。

```
x = shlr(x & 0x00FF00FF, 16) | x & ~0x00FF00FF;
x = shlr(x & 0x0F0F0F0F,  8) | x & ~0x0F0F0F0F;
x = shlr(x & 0x33333333,  4) | x & ~0x33333333;
x = shlr(x & 0x55555555,  2) | x & ~0x55555555;
x = shlr(x, 1);
```

这段代码使用“和另一个数的反码求与”（and with complement）操作来避免加载某些掩码。即使 CPU 不支持该指令，也依然有办法。可以先把第一行改写为

```
x = shlr(x, 16) & 0x00FF00FF | x & ~0x00FF00FF;
```

这样的话，可以看出它是一个“数据选择操作”（MUX operation），所以根据恒等式

$$x \,\&\, m \mid y \,\&\, \neg m = ((x \oplus y) \,\&\, m) \oplus y$$

可得

```
x = ((shlr(x, 16) ^ x) & 0x00FF00FF) ^ x;
```

带有“和另一个数的反码求与”操作的其他几行也可按类似方式改写。

还有个稍好一点的方法对很多计算机都适用，它能略微发挥 CPU 的指令集并发能力。该算法利用了等式 [Karv]

$$x \,\&\, \neg m = (x \,\&\, m) \oplus x$$

并采用常见的“与操作”表达式。函数代码如图 7.2 所示（需 17 条指令，再加 8 条常数加载指令，共计 25 条）。

```
unsigned rev(unsigned x) {
   unsigned t;
   t = x & 0x00FF00FF; x = shlr(t, 16) | t ^ x;
   t = x & 0x0F0F0F0F; x = shlr(t,  8) | t ^ x;
   t = x & 0x33333333; x = shlr(t,  4) | t ^ x;
   t = x & 0x55555555; x = shlr(t,  2) | t ^ x;
   x = shlr(x, 1);
   return x;
}
```

图 7.2 通过循环移位来反转位元

也许应该提一下，0x00FF 00FF 与 0x0F0F 0F0F 等常数可以用下列递推关系式生成。这对 32 位机没有用（甚至可能降低指令并发执行能力），因为带有 32 位 RISC 指令集的 CPU 只用两条指令就能载入这些常数。但 64 位机也许用得到下面这些式子：

$$\begin{aligned} &\boldsymbol{C}_0 \leftarrow \mathbf{0x00000000\ FFFFFFFF} \\ &\boldsymbol{C}_1 \leftarrow \boldsymbol{C}_0 \oplus (\boldsymbol{C}_0 \ll 16) \\ &\boldsymbol{C}_2 \leftarrow \boldsymbol{C}_1 \oplus (\boldsymbol{C}_1 \ll 8) \\ &\cdots \end{aligned}$$

还有一种反转位元的方法，就是把二进制数分为三组，对调左右两组，保留中间那组 [Baum]。下面演示如何反转长度为 27 位的字组。

```
012345678 9abcdefgh ijklmnopq   The given 27-bit word
ijklmnopq 9abcdefgh 012345678   First ternary swap
opqlmnijk fghcde9ab 678345012   Second ternary swap
qponmlkji hgfedcba9 876543210   Third ternary swap
```

直观的实现代码如下。若在 32 位机上运行，则该算法会反转 0 至 26 号位元，将结果

放在编号为 0 至 26 的位置中，并将第 27 至 31 位清零。

```
x = (x & 0x000001FF) << 18 | (x & 0x0003FE00) |
    (x >> 18) & 0x000001FF;
x = (x & 0x001C0E07) <<  6 | (x & 0x00E07038) |
    (x >> 6) & 0x001C0E07;
x = (x & 0x01249249) <<  2 | (x & 0x02492492) |
    (x >> 2) & 0x01249249;
```

上述算法需 21 条基本 RISC 指令，外加 10 条常数加载指令，共计 31 条。与之相对应的是，图 7.1 中的代码要 24 条基本指令，再加 6 条常数加载指令，还要用一条右移指令将结果移动 5 位，以便右对齐。因此，当待反转的位元数不超过 27 个时，"三元算法"（tenary method）和图 7.1 的算法一样好，或更优于它。

接下来这个函数由高德纳（Donald E. Knuth）先生给出 [Knu8]，这个算法很有意思，因为它反转一个 32 位字组只用 4 步，而且移位长度及掩码都极不规则，出人意料。算法用了一次循环移位操作及 3 次"三元交换"（tenary swap）操作。其步骤如下：

```
01234567 89abcdef ghijklmn opqrstuv   Given
fghijklm nopqrstu v0123456 789abcde   Rotate left 15
pqrstuvm nofghijk labcde56 78901234   10-swap
tuvspqrm nojklifg hebcda96 78541230   4-swap
vutsrqpo mnlkjihg fedcba98 76543210   2-swap
```

直观的实现代码如下。

```
x = shlr(x, 15);               // Rotate left 15.
x = (x & 0x003F801F) << 10 | (x & 0x01C003E0) |
    (x >> 10) & 0x003F801F;
x = (x & 0x0E038421) <<  4 | (x & 0x11C439CE) |
    (x >>  4) & 0x0E038421;
x = (x & 0x22488842) <<  2 | (x & 0x549556B5) |
    (x >>  2) & 0x22488842;
```

改写此算法，虽可减少指令数，但会削弱其并发能力。下面这行代码

```
x = (x & M1) << s | (x & M2) | (x >> s) & M1;
```

之中的 M2 是～（M1 |(M1≪s))。可把它改写为

```
t = (x ^ (x >> s)) & M1; x = (t | (t << s)) ^ x;
```

于是得出图 7.3 中的代码（需 19 条完整 RISC 指令，再加 6 条常数加载指令，共 25 条）。

```
unsigned rev(unsigned x) {
   unsigned t;

   x = shlr(x, 15);               // Rotate left 15.
   t = (x ^ (x>>10)) & 0x003F801F; x = (t | (t<<10)) ^ x;
   t = (x ^ (x>> 4)) & 0x0E038421; x = (t | (t<< 4)) ^ x;
   t = (x ^ (x>> 2)) & 0x22488842; x = (t | (t<< 2)) ^ x;
   return x;
}
```

图 7.3 用 Knuth 算法反转位元

如果可以用循环移位指令，那么在反转 32 位二进制数时，Knuth 算法的指令数并不比图 7.2 中的算法（17 条指令外加 8 条常数加载指令）少，不过，Knuth 算法的代码只

用了一次循环移位指令。如果把这条指令改写成

```
x = (x << 15) | (x >> 17);   // Rotate left 15.
```

那么 Knuth 算法只需 21 条基本 RISC 指令，外加 6 条常数加载指令，如果仅用基本 RISC 指令来反转 32 位字组，那么它就是最佳算法。这促使我们思考，有没有简单方法能预判反转某个定长二进制数所需的移位及逻辑操作次数呢？

是否可将 Knuth 算法扩展至 64 位机，以此反转 64 位二进制数呢？可以，有两种方法，一种简单，一种复杂。简单的方法是先把 64 位寄存器左右两半交换，然后运用 32 位版 Knuth 算法并行处理这两部分。图 7.4 描述了其代码。若把交换操作（也就是循环移动 32 位）算作一条指令，则共需 24 条。

```
unsigned long long rev(unsigned long long x) {
   unsigned long long t;

   x = (x << 32) | (x >> 32);    // Swap register halves.
   x = (x & 0x0001FFFF0001FFFFLL) << 15 | // Rotate left
       (x & 0xFFFE0000FFFE0000LL) >> 17;  // 15.
   t = (x ^ (x >> 10)) & 0x003F801F003F801FLL;
   x = (t | (t << 10)) ^ x;
   t = (x ^ (x >> 4)) & 0x0E0384210E038421LL;
   x = (t | (t << 4)) ^ x;
   t = (x ^ (x >> 2)) & 0x2248884222488842LL;
   x = (t | (t << 2)) ^ x;
   return x;
}
```

图 7.4　64 位版 Knuth 算法

另外一种方法是仿照 32 位版 Knuth 算法，找到合适的移位量及掩码。该算法代码如下。若将循环左移 31 位视为一条指令，则共需 25 条。

```
unsigned long long rev(unsigned long long x) {
   unsigned long long t;

   x = (x << 31) | (x >> 33);   // I.e., shlr(x, 31).
   t = (x ^ (x >> 20)) & 0x00000FFF800007FFLL;
   x = (t | (t << 20)) ^ x;
   t = (x ^ (x >> 8)) & 0x00F8000F80700807LL;
   x = (t | (t << 8)) ^ x;
   t = (x ^ (x >> 4)) & 0x0808708080807008LL;
   x = (t | (t << 4)) ^ x;
   t = (x ^ (x >> 2)) & 0x1111111111111111LL;
   x = (t | (t << 2)) ^ x;
   return x;
}
```

也可以用查表法来改进位元反转算法。下列代码每次从 256 字节的表中查出一个字节，然后把 4 次查表结果反向连缀起来。如果把循环展开成顺序代码，那么共需 13 条基本 RISC 指令，外加 4 条载入指令，这样看来，在某些计算机上，它也许是最佳算法。

```
unsigned rev(unsigned x) {
   static unsigned char table[256] = {0x00, 0x80, 0x40,
   0xC0, 0x20, 0xA0, 0x60, 0xE0, ..., 0xBF, 0x7F, 0xFF};
   int i;
   unsigned r;
```

```
   r = 0;
   for (i = 3; i >= 0; i--) {
      r = (r << 8) + table[x & 0xFF];
      x = x >> 8;
   }
   return r;
}
```

7.1.1 位元反转算法的推广

Steele先生提议，下述广义位元反转操作可纳入计算机指令集中，谓之“翻转”(flip)指令[GLS1]：

```
if (k &  1) x = (x & 0x55555555) <<  1 | (x & 0xAAAAAAAA) >>  1;
if (k &  2) x = (x & 0x33333333) <<  2 | (x & 0xCCCCCCCC) >>  2;
if (k &  4) x = (x & 0x0F0F0F0F) <<  4 | (x & 0xF0F0F0F0) >>  4;
if (k &  8) x = (x & 0x00FF00FF) <<  8 | (x & 0xFF00FF00) >>  8;
if (k & 16) x = (x & 0x0000FFFF) << 16 | (x & 0xFFFF0000) >> 16;
```

(最后两行的“与”操作可省去。)若k=31，则用上述操作即可反转字组。若k=24，则可反转字组中的字节顺序。若k=7，则只反转每个字节内部的位元，而不改变字节之间的顺序。若k=16，则反转字组左右两部分。余可类推。总之，它把位置m上的位元移到位置$m \oplus k$。该操作的硬件实现方式与循环移位指令的常见实现方式极为相似(都是5阶段的数据选择器(MUX)，每阶段都由移位量k中的一个位元所控制)。

7.1.2 奇特的位元反转算法

[HAK]的Item 167里面有几个奥妙的表达式，可以反转6位、7位与8位整数。虽说它们是为36位机设计的，不过反转6位整数的算法也可用于32位机，反转7位、8位整数的算法也可用于64位机。这些表达式分别为：

$$6\text{位：}\quad \mathrm{remu}((x * \mathbf{0x\ 0008\ 2082})\ \&\ \mathbf{0x0112\ 2408}, \mathbf{255})$$

$$7\text{位：}\quad \mathrm{remu}((x * \mathbf{0x\ 4010\ 0401})\ \&\ \mathbf{0x4\ 4221\ 1008}, \mathbf{255})$$

$$8\text{位：}\quad \mathrm{remu}((x * \mathbf{0x2\ 0202\ 0202})\ \&\ \mathbf{0x108\ 8442\ 2010}, \mathbf{1023})$$

上述算法的结果都是“干净”整数(“clean” integer)：相关位元已经右对齐，无用的高权重位元全部清零。

这些式子中的remu函数都可以用rem或mod替换，因为参数总是正数。使用求余函数(remainder function)只是想把其操作视为256进制或1024进制数，并统计每一数位上的数字和而已，这与“舍九法”(casting out nines)[⊖]很相似。因此，可以用乘法及右移操作代替。例如，在32位机上可用下列算法取代6位整数反转公式(乘法结果必须和2^{32}取模)：

$$t \leftarrow (x * \mathbf{0x00082082})\ \&\ \mathbf{0x01122408}$$

$$(t * \mathbf{0x01010101}) \overset{u}{\gg} 24$$

⊖ 又叫“去九法”，是一种通过统计数字中的数位和来验算加、减、乘、除的方法。详情参见：https://en.wikipedia.org/wiki/Casting_out_nines。——译者注

这些公式的用途有限，因为它们要执行求余操作（至少要 20 个周期）和/或某些乘法操作，而且还得载入“大常数”。上面刚写的这个算法需要 10 条基本 RISC 指令，其中两条是乘法指令，这种指令在当前具备 RISC 指令集的 CPU 中大约要 20 个周期。另一方面，采用图 7.1 中的代码来反转 6 位整数大概要 15 条指令，根据计算机的指令并发执行能力，可能要用大约 9～15 个周期。虽然执行速度慢，但是用这些位元反转技巧确实写出了精练的代码。下面再讲一些 32 位机也许能用到的技巧。它们都将［HAK］所述的思路用了两次，以便在 32 位机上反转 8 位及 9 位整数。

下列公式可反转 8 位整数：

$$
\begin{aligned}
&s \leftarrow (x * \mathbf{0x0202\ 0202})\ \&\ \mathbf{0x8442\ 2010}\\
&t \leftarrow (x * \mathbf{8})\ \&\ \mathbf{0x0000\ 0420}\\
&\mathrm{remu}(s+t, \mathbf{1023})
\end{aligned}
$$

这里的 remu 无法改用乘法加移位替换。（其中的道理请读者自己探究，看看二进制位的模式（bit pattern），找找原因。）

下面还有一个类似的公式，也能反转 8 位整数。此公式值得研究，因为它可以被极大地简化：

$$
\begin{aligned}
&s \leftarrow (x * \mathbf{0x0002\ 0202})\ \&\ \mathbf{0x0104\ 4010}\\
&t \leftarrow (x * \mathbf{0x0008\ 0808})\ \&\ \mathbf{0x0208\ 8020}\\
&\mathrm{remu}(s+t, \mathbf{4095})
\end{aligned}
$$

可以简化的原因是：第二个乘积可由第一个乘积左移后得到；依照前一个掩码，即可由一条移位指令生成后一个掩码；求余运算可替换为乘法及移位操作。这样就简化成 14 条基本 RISC 指令了，其中两条为乘法：

$$
\begin{aligned}
&u \leftarrow x * \mathbf{0x00020202}\\
&m \leftarrow \mathbf{0x01044010}\\
&s \leftarrow u\ \&m\\
&t \leftarrow (u \ll 2)\ \&(m \ll 1)\\
&(\mathbf{0x01001001} * (s+t)) \overset{u}{\gg} 24
\end{aligned}
$$

下列公式可反转 9 位整数：

$$
\begin{aligned}
&s \leftarrow (x * \mathbf{0x0100\ 1001})\ \&\ \mathbf{0x8410\ 8010}\\
&t \leftarrow (x * \mathbf{0x0004\ 0040})\ \&\ \mathbf{0x0084\ 1080}\\
&\mathrm{remu}(s+t, \mathbf{1023})
\end{aligned}
$$

第二个乘法运算可以省去，因为其结果等于第一次乘法之积右移 6 位。第二个掩码等于第一个掩码右移 8 位。执行上述化简之后，算法需要 12 条基本 RISC 指令，其中含有一条乘法指令及一条求余指令。求余操作必须是无符号版，而且不能用乘法及移位操作代替。

读者研究了这些奇妙算法后，也可以自己用代码实现一些类似的位元排列操作（bit

permuting operation）。笔者刻意构造了一个简单的例子：假设要从某个8位二进制数中每隔一个位置取一个位元，并把4个结果位元靠右对齐。也就是说，我们想实现如下变换：

```
0000 0000 0000 0000 0000 0000 abcd efgh ==>
0000 0000 0000 0000 0000 0000 0000 bdfh
```

此操作可按如下方式计算：

$$t \leftarrow (x * \mathbf{0x01010101}) \,\&\, \mathbf{0x40100401}$$

$$(t * \mathbf{0x08040201}) \overset{u}{\gg} 27$$

在大多数计算机中，要执行上述各种操作，最实际的办法还是通过索引查表。表中元素可以是单字节整数或9位二进制整数。

7.1.3 递增反转后的整数

[PuBr] 所述的“快速傅里叶变换”[1]（Fast Fourier Transform，FFT）算法中有个循环，其中用到了一个整数 i 及其位元反转值 rev（i），i 值每次随循环递增1。若 i 值较小，则查表法尚且实用，因为它很快就能算出 rev（i）。但如果遇上“大整数”，那么查表法就不太实用了，因为可能需要29条基本RISC指令才行[2]。

如果不能用查表法，那么可于每次循环内同时计算新的 i 以及该数反转位元后的值，这样效率更高。问题是，假如寄存器中已经有某数的位元反转值了，现在想直接据此推算出下一个数的位元反转值，用何种算法好呢？[3]下面演示在4位机上需要算出的一连串步进制（用十六进制表示）：

```
0, 8, 4, C, 2, A, 6, E, 1, 9, 5, D, 3, B, 7, F.
```

在FFT算法中，i 及其位元反转值都是某个定长的二进制数，几乎可以肯定其位元数都小于32，并且均在寄存器中靠右对齐。不过，本书假设 i 就是32位二进制整数。将反转后的32位二进制数加1然后右移，这样得出的值就可适用于FFT算法了（i 与 rev(i)都用作下标，以访问内存中的数组）。

直观的计算方法是，从左侧扫描位元反转后的值，直至发现“0”，将其设为1，并把它左方全部位元（如果有的话）置0，这样就可直接得出下一个数的位元反转值了。该算法的一种实现代码是：

---

[1] 离散傅里叶变换的快速算法也可计算离散傅里叶变换的逆变换。快速傅里叶变换广泛用于数字信号处理、大整数乘法、偏微分方程求解等领域。详情参见：https：//zh.wikipedia.org/wiki/快速傅里叶变换。——译者注

[2] 29条基本RISC指令，指用查表法反转64位元整数，并把循环代码展开后所需的指令数，不含加载指令。——译者注

[3] “递增”通常指“5加1变成6、6加1变成7”这种递进增长，而本节标题中的“递增”一词，却是指根据“5”的位元反转值直接推算出其下一个数，也就是“6”的位元反转值，故译文加引号，以强调其特殊含义。——译者注

```
unsigned x, m;

m = 0x80000000;
x = x ^ m;
if ((int)x >= 0) {
   do {
      m = m >> 1;
      x = x ^ m;
   } while (x < m);
}
```

如果 x 首个位元是“0”，那么上述代码需要 3 条基本指令，否则就要继续执行循环，而每次循环又需要 4 条指令。由于 x 以“0”开头的几率是 1/2，以二进制“10”开头的几率是 1/4，以此类推，该算法的平均指令数约为：

$$
\begin{aligned}
&3\cdot\frac{1}{2}+7\cdot\frac{1}{4}+11\cdot\frac{1}{8}+15\cdot\frac{1}{16}+\ldots\\
&=4\cdot\frac{1}{2}+8\cdot\frac{1}{4}+12\cdot\frac{1}{8}+16\cdot\frac{1}{16}+\ldots-1\\
&=4\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\ldots\right)-1\\
&=7
\end{aligned}
$$

给上述算式第一行加 1 再减 1，就得到了第二行。其中加上去的“1”写成了 1/2＋1/4＋1/8＋1/16＋… 的形式，这和 5.4 节分析二进制数中后缀 0 的平均个数时所用数列类似。不过，在最差情况下，此算法需要多达 131 条指令数。

若能使用“前导 0 计数指令”，则可用下列方法算出某数反转位元后再加 1 的值：

$$\text{首先执行：}\boldsymbol{s} \leftarrow \mathrm{nlz}(\neg \boldsymbol{x})$$

$$\text{然后执行：}\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus (\mathbf{0x80000000} \overset{s}{\gg} \boldsymbol{s})$$

$$\text{或：}\boldsymbol{x} \leftarrow ((\boldsymbol{x} \ll \boldsymbol{s}) + \mathbf{0x80000000}) \overset{u}{\gg} \boldsymbol{s}$$

上述两种方法都需要 5 条完整的 RISC 指令，而且，为了使 0xFFFF FFFF 再加 1 能折回到 0，必须采用“模 64”的移位操作。(由此可见，这些公式在 Intel x86 计算机中不成立，因为其移位操作是“模 32”的。)

下面这行令人费解的算法只需 6 条基本 RISC 指令，即可由某数的位元反转值推算出下一个数的位元反转值 [Möbi]。该算法无分支和加载操作，但却需要一个除法操作。它可处理的二进制数长度比计算机的字组长度少 1。

$$\boldsymbol{revi} \leftarrow \boldsymbol{revi} \oplus \left(\boldsymbol{m} - \frac{\boldsymbol{m}}{(\boldsymbol{i} \oplus (\boldsymbol{i}+\mathbf{1}))+\mathbf{1}}\right)$$

必须同时知道未反转的 $\boldsymbol{i}$ 及反转后的 $\boldsymbol{revi}$ 方可使用此算法。变量 $\boldsymbol{m}$ 是基准数值，如果待处理整数有 n 个位元，则 $\boldsymbol{m}=2^n$。运用此公式即可得出下一个整数的位元反转值。未反转的整数 $\boldsymbol{i}$ 可自行递增。该算法“就地”(“in place”) 算出下一个数的位元反转值，也就是说，与前两种算法不同，它并不会先把算好的值放在寄存器中权重高的一端，然后再向右对齐，而是直接到位。

此算法有种变体：

$$revi \leftarrow revi \oplus \left(m - \frac{m/2}{\neg i \ \& \ (i+1)}\right) \tag{1}$$

如果 m 是常数，那么就不用再计算 $m/2$ 了，因为它也是常数。与此同时，若计算机还支持“and not”指令，则此算法共需5条指令。它支持的二进制数长度与计算机的字组长度相等。（如果要计算“全字长”整数（full word-size integer）[㊀]，那么公式中第一个 m 用0替换，后面的 $m/2$ 用 2^{n-1} 替换。）

7.2 乱序排列位元

还有一种重要的字组位元排列法，就是“完美洗牌”操作（“perfect shuffle” operation）[㊁]，它可用于密码学（cryptography）。该算法有两个变体，分别叫做“外完美洗牌”（outer perfect shuffle）和“内完美洗牌”（inner perfect shuffle）[㊂]。这两种变体算法都把字组平分成两组，然后把其中一组位元交错地插在另一组位元中，这与32张牌的“完美洗牌法”类似，两者的区别在于哪一堆牌插在前面。在“外完美洗牌”算法中，左边那堆牌最上面一张（也就是最外面那张牌）洗完牌之后仍然在最上方，而“内完美洗牌”算法则把洗牌前的15号位元移至洗牌后的字组最左端（也就是31号位置）。假设有下列32位字组（其中每个字母表示一个二进制位）：

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP
```

“外完美洗牌”的结果是：

```
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP
```

而“内完美洗牌”的结果是：

```
AaBb CcDd EeFf GgHh IiJj KkLl MmNn OoPp
```

假定字宽 W 是2的幂。那么可以分 $\log2(W/2)$ 个步骤，以基本RISC指令实现出“外完美洗牌”操作。第一步会把当前所有位元视为一组，其后每步都会把前一步中的组平分为两个新组。每一步均要将当前各组内的位元划成四等份，交换第二、三等份[GLS1]。也就是说，32位字组将按下列步骤变换：

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP
abcd efgh ABCD EFGH ijkl mnop IJKL MNOP
```

㊀ 也就是其位宽与计算机字组长度相同的整数，在32位机上就是32位二进制整数，在64位机上就是64位二进制整数。——译者注

㊁ 也叫Faro shuffle，详情参见：https://en.wikipedia.org/wiki/Faro_shuffle。——译者注

㊂ 两种变体分别参见：https://en.wikipedia.org/wiki/Out_shuffle 及 https://en.wikipedia.org/wiki/In_shuffle。——译者注

```
abcd ABCD efgh EFGH ijkl IJKL mnop MNOP
abAB cdCD efEF ghGH ijIJ klKL mnMN opOP
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP
```

直观的实现代码是：

```
x = (x & 0x0000FF00) << 8 | (x >> 8) & 0x0000FF00 | x & 0xFF0000FF;
x = (x & 0x00F000F0) << 4 | (x >> 4) & 0x00F000F0 | x & 0xF00FF00F;
x = (x & 0x0C0C0C0C) << 2 | (x >> 2) & 0x0C0C0C0C | x & 0xC3C3C3C3;
x = (x & 0x22222222) << 1 | (x >> 1) & 0x22222222 | x & 0x99999999;
```

这需要 42 条基本 RISC 指令。若改用“异或法”来交换寄存器中的两个位段（参见 2.20.2 节），则可减少至 30 条，不过，在指令并发执行能力不受限的计算机中，其执行周期也会由原来的 17 个上升为 21 个。其中所有数均无符号。

```
t = (x ^ (x >> 8)) & 0x0000FF00;  x = x ^ t ^ (t << 8);
t = (x ^ (x >> 4)) & 0x00F000F0;  x = x ^ t ^ (t << 4);
t = (x ^ (x >> 2)) & 0x0C0C0C0C;  x = x ^ t ^ (t << 2);
t = (x ^ (x >> 1)) & 0x22222222;  x = x ^ t ^ (t << 1);
```

将上述步骤反向执行，即可实现逆操作，也就是“外完美理牌”（outer perfect unshuffle）[⊖]算法。

```
t = (x ^ (x >> 1)) & 0x22222222;  x = x ^ t ^ (t << 1);
t = (x ^ (x >> 2)) & 0x0C0C0C0C;  x = x ^ t ^ (t << 2);
t = (x ^ (x >> 4)) & 0x00F000F0;  x = x ^ t ^ (t << 4);
t = (x ^ (x >> 8)) & 0x0000FF00;  x = x ^ t ^ (t << 8);
```

就上述两种洗牌算法来说，如果只执行最后两步，那么仅会在字组中每个字节内部分别乱序排列其位元，而不会影响其他字节。如果只执行最后三步，那么仅会在字组的左右两个半字组内部分别乱序排列其位元，而不会影响另一侧的半字组。理牌算法也如此，只是要把刚才说的“最后两步”、“最后三步”改成“头两步”、“头三步”。

要实现“内完美洗牌”，只需在“外完美洗牌”的代码前加一步，交换寄存器左右两半的内容即可：

```
x = (x >> 16) | (x << 16);
```

（或者循环移动 16 个位元。）“内完美理牌”算法也类似，把上面这句代码加在“外完美理牌”算法后面即可。

把“内完美洗牌”算法每一步内的交换方式由第二、三等份改为第一、四等份，而每一步的分组方式不变，依然是把上一步的组平分为两个新组，这样的话，用新算法求出的数就是“内完美洗牌”算法结果的位元反转值。

有个特例值得注意，就是字组 x 的左半字均为“0”。换句话说，就是要把右半字中的位元交错嵌入其中，这样的话，原来的 32 位字组

```
0000 0000 0000 0000 ABCD EFGH IJKL MNOP
```

会变换成

⊖ unshuffle 也称“逆洗牌”、“解混洗”、“逆混洗”。——译者注

```
0A0B 0C0D 0E0F 0G0H 0I0J 0K0L 0M0N 0O0P.
```

“外完美洗牌”的代码此时可简化成22条基本RISC指令。不过下面列出的这种代码只用了19条，在指令并发执行能力不受限的计算机上，它和直接由“外完美洗牌”算法简化而成的代码相比，执行时间并未增加（两者都要执行12个周期）。执行代码前不需要清空x的左半字。

```
x = ((x & 0xFF00) << 8) | (x & 0x00FF);
x = ((x << 4) | x) & 0x0F0F0F0F;
x = ((x << 2) | x) & 0x33333333;
x = ((x << 1) | x) & 0x55555555;
```

同理，这种“半洗牌”操作的逆算法（其为7.4节“压缩”算法的特例），也就是“外完美半理牌”算法，可以简化成29或26条基本RISC指令，具体是哪种情况要取决于执行算法前是否需要先用“与”操作把奇数位置上的位元清零。不过，下列代码只需18或21条基本RISC指令，在指令并发执行能力不受限的计算机中，它和直接由“外完美理牌”算法简化而成的代码相比，执行时间更短（只需15或12个周期）。

```
x = x & 0x55555555;          // (If required.)
x = ((x >> 1) | x) & 0x33333333;
x = ((x >> 2) | x) & 0x0F0F0F0F;
x = ((x >> 4) | x) & 0x00FF00FF;
x = ((x >> 8) | x) & 0x0000FFFF;
```

7.3 转置位矩阵

把矩阵A的行变为另一个矩阵的列，把矩阵A的列变为另一个矩阵的行，这个新矩阵就是A的转置矩阵。本节研究如何计算位矩阵的转置矩阵。位矩阵中的元素都是位元，每8个位元打包为一个字节，行与列都始于字节边界（byte boundary）[⊖]。这个看似简单的变换操作却需要耗费很多指令。

在大多数计算机上加载或保存单个位元都很慢，这主要是由于提取或存储单个位元所需的代码而致（存储比提取更糟）。好一些的方法是把矩阵划分为8×8的子矩阵，然后将每个8×8的子矩阵载入寄存器，在其中算出它的转置子矩阵，然后把8×8的转置结果放到目标矩阵的适当位置上。图7.5演示了如何转置2×3字节的位矩阵。A，B，…，F都是8×8位元的子矩阵，而A^T，B^T，… 则分别代表A，B，… 的转置子矩阵。

在转置8×8子矩阵时，位矩阵的存储顺序是“行优先”（row-major）还是“列优先”（column-major）都可以，两种情况下的操作方法均相同。本书假定矩阵存储顺序是“行优先”。这样的话，矩阵首字节就是子矩阵A的顶行，次字节就是子矩阵B的顶行，以此类推。若L表示某个子矩阵的首字节（也就是其最上方那一行）地址，则该子矩阵其后各行的地址就是$L+n$，$L+2n$，…，$L+7n$。

⊖ 通俗地说，就是每行、每列的位元个数都是8的倍数。——译者注

研究此问题时，我们抛开原来一贯采用的假定，不再使用 32 位计算机了，而是假设计算机带有 64 位通用寄存器。这样，算法就更简单、更易理解了，而且也不难把它转换为可在 32 位机上执行的程序。实际上，只要 32 位机的编译器支持 64 位整数操作就行（虽说这么编译出来的代码也许不及手工改写的代码高效）。

大思路是，先用 8 条“字节加载”指令把子矩阵中的字节从左至右规整到 64 位通用寄存器中。然后根据寄存器中的内容计算转置矩阵。最后，再用 8 条“字节存储”指令把结果保存到目标区域。

下图演示了如何转置 8×8 的位矩阵，其中每个字符表示一个位元。

```
0123 4567          08go wEMU
89ab cdef          19hp xFNV
ghij klmn          2aiq yGOW
opqr stuv    ⇒     3bjr zHPX
wxyz ABCD          4cks AIQY
EFGH IJKL          5dlt BJRZ
MNOP QRST          6emu CKS$
UVWX YZ$.          7fnv DLT.
```

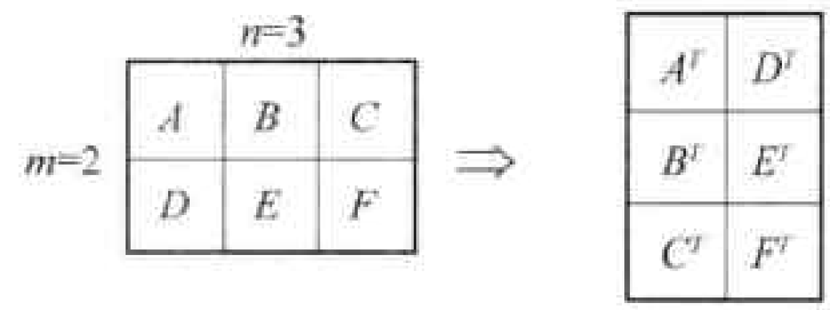


图 7.5 转置 16×24 的位矩阵

如果把全部位元视为双字，那么变换过程就是把第一行变为第二行。

```
01234567 89abcdef ghijklmn opqrstuv wxyzABCD EFGHIJKL MNOPQRST UVWXYZ$.
08g0wEMU 19hpxFNV 2aiqyGOW 3bjrzHPX 4cksAIQY 5dltBJRZ 6emuCKS$ 7fnvDLT.
```

可以看到，字符“1”所代表的位元向右移动了 7 个位置，字符“2”所代表的位元向右移动了 14 个位置，字符“8”所代表的位元向左移动了 7 个位置。每个位元都会向左或向右移动 0、7、14、28、35、42 或 49 个位置。由于双字里共有 56 个位元待移动，而只有 14 种不同的移位量，所以可选取适当的掩码及移位方式，平均每次移动 4 个位元。直观的实现代码如下：

```
y =  x & 0x8040201008040201LL              |
    (x & 0x0080402010080402LL) <<  7 |
    (x & 0x0000804020100804LL) << 14 |
    (x & 0x0000008040201008LL) << 21 |
    (x & 0x0000000080402010LL) << 28 |
    (x & 0x0000000000804020LL) << 35 |
    (x & 0x0000000000008040LL) << 42 |
    (x & 0x0000000000000080LL) << 49 |
    (x >>  7) & 0x0080402010080402LL |
    (x >> 14) & 0x0000804020100804LL |
    (x >> 21) & 0x0000008040201008LL |
    (x >> 28) & 0x0000000080402010LL |
    (x >> 35) & 0x0000000000804020LL |
    (x >> 42) & 0x0000000000008040LL |
    (x >> 49) & 0x0000000000000080LL;
```

上述代码共需执行 43 条基本 RISC 指令，其中不计生成掩码所需的指令（由于掩码都是反复出现的常数，所以在转置大的位矩阵时，此类指令不重要）。循环移位操作无助于实现此算法。有些项具备 (x & mask)≪s 的形式，还有些是 (x≫s) & mask 的形式。这减少了算法所用的掩码种数：最后 7 个掩码分别重复了前面 7 个。可以看出，第一个掩码之后的各掩码都可通过右移位指令由第一个掩码生成。这样的话，很容易就能用一个执行 7 次的 for 循环写出更精简的版本来。

另一种变体是通过 Steele 先生说的“异或法”来交换位段（参见 2.20.2 节），不过这

个技巧也无法显著改善此计算过程。不算掩码生成指令，用它写出来的函数需要执行42条指令。其代码以

```
t = (x ^ (x >> 7)) & 0x0080402010080402LL;
x = x ^ t ^ (t << 7);
```

起头，共需7组这样的语句。

虽说看上去似乎没有太好的算法能解决此问题，不过接下来要介绍的这个方法比刚才那种直白的算法和它的各种变体都强，在具备基本RISC指令集的计算机上，其计算部分（不含载入、存储子矩阵及生成掩码）的效率约为上述算法的两倍。此算法得力于其高度的“位并行”（bit-parallelism）能力。当矩阵元素都为字组而非位元时，它可能称不上一个好算法。在那种情况下，最好还是直接载入每个字组并把它存放到正确位置上。

首先，把8×8的位矩阵划分为16个2×2的位矩阵，分别转置这16个2×2的矩阵。然后把整个矩阵划分成4个2×2的子矩阵，每个子矩阵的元素是2×2的位矩阵，转置这4个2×2的子矩阵。最后，把整个矩阵视为一个2×2矩阵，其元素是4×4的位矩阵，转置这个2×2的矩阵。变换过程如下所示［Floyd］：

```
0123 4567     082a 4c6e     08go 4cks     08go wEMU
89ab cdef     193b 5d7f     19hp 5dlt     19hp xFNV
ghij klmn     goiq ksmu     2aiq 6emu     2aiq yGOW
opqr stuv  ⇒  hpjr ltnv  ⇒  3bjr 7fnv  ⇒  3bjr zHPX
wxyz ABCD     wEyG AICK     wEMU AIQY     4cks AIQY
EFGH IJKL     xFzH BJDL     xFNV BJRZ     5dlt BJRZ
MNOP QRST     MUOW QYS$     yGOW CKS$     6emu CKS$
UVWX YZ$.     NVPX RZT.     zHPX DLT.     7fnv DLT.
```

完整算法列在图7.6中。参数A表示8×8子矩阵的首字节在源矩阵中的地址，而参数B则是8×8子矩阵的首字节在目标矩阵中的地址。

```
void transpose8(unsigned char A[8], int m, int n,
                unsigned char B[8]) {
   unsigned long long x;
   int i;

   for (i = 0; i <= 7; i++)      // Load 8 bytes from the
      x = x << 8 | A[m*i];       // input array and pack
                                 // them into x.

   x =  x & 0xAA55AA55AA55AA55LL             |
       (x & 0x00AA00AA00AA00AALL) <<  7 |
       (x >> 7) & 0x00AA00AA00AA00AALL;
   x =  x & 0xCCCC3333CCCC3333LL             |
       (x & 0x0000CCCC0000CCCCLL) << 14 |
       (x >> 14) & 0x0000CCCC0000CCCCLL;
   x =  x & 0xF0F0F0F00F0F0F0FLL             |
       (x & 0x00000000F0F0F0F0LL) << 28 |
       (x >> 28) & 0x00000000F0F0F0F0LL;

   for (i = 7; i >= 0; i--) {    // Store result into
      B[n*i] = x; x = x >> 8;}   // output array B.
}
```

图7.6 转置8×8的位矩阵

该函数计算部分共需21条指令。由于三大步骤中执行的都是调换位元操作，所以可

用 Steele“异或法”来交换位段。这样的话，图 7.6 的第一个赋值语句就变为：

```
t = (x ^ (x >> 7)) & 0x00AA00AA00AA00AALL;
x = x ^ t ^ (t << 7);
```

算法修改之后，其计算部分只需 18 条指令，不过几乎无法发挥 CPU 的指令级并发执行能力了。

根据每次调换的位元分组大小，图 7.6 所列算法的粒度既可由细到粗，又可由粗到细。若想由粗到细执行，可以先把整个 8×8 的位矩阵视为 2×2 矩阵，其中每个元素都是 4×4 的位矩阵，转置这个 2×2 的矩阵。然后，把每个 4×4 子矩阵当成 2×2 矩阵，其中每个元素是 2×2 的位矩阵，分别转置这 4 个 2×2 的子矩阵，以此类推。这样写成的代码与图 7.6 一样，只是那三条用于重排位元的赋值语句的顺序颠倒了。

正如前文所述，用两个 32 位寄存器来表示 64 位量并修改代码，就能令上面那些函数在 32 位机上执行。写出代码，化简运算结果显然为 0 的操作后，32 位版直白算法需要 74 条指令（与之相比，64 位计算机所用算法只需 43 条），而改为 32 位版的图 7.6 需要 36 条指令（64 位版需 21 条）。在 64 位计算机上，也可以用“Steele 位元交换技巧”来减少指令数，不过要损失指令并发能力。

转置 32×32 的位矩阵

这种递归技巧也可用于转置大于 8×8 的位矩阵。转置 32×32 的位矩阵需要分 5 个阶段。

实现细节与图 7.6 很不相同：因为此处假定通用寄存器放不下整个 32×32 位矩阵，所以还要想个简洁的办法，以便在位矩阵中找到需要调换其位元的字组。接下来要讲的这个算法从粗粒度到细粒度执行时，效果最好。

第一阶段，把矩阵视为 4 个 16×16 的位矩阵，并按如下方式变换：

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} A & C \\ B & D \end{bmatrix}$$

A 表示矩阵前 16 个字组的左半边，B 表示矩阵前 16 个字组的右半边，C、D 可类推[⊖]。显然，上述变换可分如下步骤执行：

对调 0 号字组右半边与 16 号字组左半边，
对调 1 号字组右半边与 17 号字组左半边，
……
对调 15 号字组右半边与 31 号字组左半边。

⊖ 由于 A、B 都是 16×16 的位矩阵，所以其每一行都各有 16 个位元，这样刚好用一个 32 位元字组横向贯穿 A 与 B 的一行，从上到下 16 行，共计 16 个字组。作者把它们称为“前 16 个字组”，也就是 0 至 15 号字组；而 C、D 所占字组，也就是 16 至 31 号字组，则成了后“16 个字组”。整个大矩阵共 1024（32×32）个位元，用 32 个字组表示。——译者注

为了用代码实现这个过程，需要设定下标 k，令其由 0 变至 15。在由 k 所控制的循环内，把 k 号字组右半边与 $k+16$ 号字组左半边对调。

第二阶段，把矩阵划分成 16 个 8×8 的位矩阵，并按如下方式变换：

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \Rightarrow \begin{bmatrix} A & E & C & G \\ B & F & D & H \\ I & M & K & O \\ J & N & L & P \end{bmatrix}$$

上述变换可以分步执行：

将 0 号字组中由 0x00FF 00FF 所标注的位元与 8 号字组中由 0xFF00 FF00 所标注的位元对调[⊖]，

将 1 号字组中由 0x00FF 00FF 所标注的位元与 9 号字组中由 0xFF00 FF00 所标注的位元对调，余可类推。

这意味着 0 号字组中右起第 0 至 7 号位元（也就是权重最低的 8 个位元）要和 8 号字组中右起第 8 至 15 个位元对调，其余各步也如此。在每次交换操作中，第一个字组所对应的下标 k 分别是 0、1、2、3、4、5、6、7、16、17、18、19、20、21、22、23。下面这个式子可根据当前 k 值算出步进后的新 k 值：

$$k' = (k+9) \,\&\, \neg 8$$

在由 k 所控制的循环中，k 号字组中的位元要和 $k+8$ 号字组中的位元交换。

同理，第三阶段的步骤是：

把 0 号字组中由 0x0F0F 0F0F 所标注的位元同 4 号字组中由 0xF0F0 F0F0 所标注的位元互换，

把 1 号字组中由 0x0F0F 0F0F 所标注的位元同 5 号字组中由 0xF0F0 F0F0 所标注的位元互换，

余可类推。

每次交换操作中，第一个字组所对应的下标 k 分别是 0、1、2、3、8、9、10、11、16、17、18、19、24、25、26、27。用下列式子，可根据当前 k 值算出步进后的下一个 k 值：

$$k' = (k+5) \,\&\, \neg 4.$$

在由 k 所控制的循环中，k 号字组中的位元要和 $k+4$ 号字组中位元的互换。

上述算法可写成相当精简的 C 语言函数，如图 7.7 所示 [GLS1]。外层循环控制五大阶段，其下标 j 分别为 16、8、4、2、1。每轮循环也会步进掩码 m，依次将其值设为 0x0000 FFFF、0x00FF 00FF、0x0F0F 0F0F、0x3333 3333、0x5555 5555。（步进时用的小窍门是：m = m^(m<<j)。该代码没有逆运算，所以前面说此算法最好能从粗到细执行，主要原因就在这里。）内层循环步进上面说过的 k 值。其循环体将 a[k] 中由掩码 m 标识

⊖ 实际上就是把 0 号字组中左起第 1 个及第 3 个字节与 8 号字组里左起第 0 个及第 2 个字节对调。掩码中某个位置的二进制值是“1”，就表明要对调此位置上的位元。——译者注

的位元与 a[k + j] 右移 j 位后再由掩码 m 所标识的位元交换，而后者也就等同于 a[k + j] 里面由掩码 m 的反码所标识的位元。执行这些交换操作时，用到了 2.20.1 节的表格里 (c) 栏所列的"三次异或"(three exclusive or) 技巧。

在一台与基本 RISC 架构非常相似的计算机中以 GNU C 编译器编译此函数，可得 31 条指令，其中 20 条在内层循环，另有 7 条属于外层循环，但不在内层循环中。这样的话，函数共需执行 4+5(7+16・20)=1639 条指令。与之相对，如果把图 7.6 中的 8×8 转置程序（改写为 32 位版）调用 16 次，那么也能实现此函数的功能，不过它需要 16(101+5) =1696 条指令（假设这 16 次调用是依次排开，而不是以循环方式执行）。其中的"5"是指每次调用函数都要花 5 条指令（由编译后的代码可知这一点）。因此，两种方法从表面上看，其执行时间相当接近。

另一方面，只要稍微修改图 7.7 的代码，即可在 64 位机上转置 64×64 的位矩阵，那样大概需要 4+6(7+32・20)=3886 条指令。执行 64 次 8×8 位矩阵转置法需要大约 64(85+5)=5760 条指令。

```
void transpose32(unsigned A[32]) {
   int j, k;
   unsigned m, t;

   m = 0x0000FFFF;
   for (j = 16; j != 0; j = j >> 1, m = m ^ (m << j)) {
      for (k = 0; k < 32; k = (k + j + 1) & ~j) {
         t = (A[k] ^ (A[k+j] >> j)) & m;
         A[k] = A[k] ^ t;
         A[k+j] = A[k+j] ^ (t << j);
      }
   }
}
```

图 7.7　转置 32×32 位矩阵所用的精简代码

此算法会在原矩阵上直接修改数据 (work in place)，所以如果要用它转置大矩阵，那么还要用附加步骤来移动 32×32 位的子矩阵。我们可以单独处理第一次或最后一次下标为 j 的 for 循环，将该次循环的结果保存到不同于源矩阵的另一块区域中，这样就能把目标矩阵与源矩阵区隔开了。

图 7.7 中，有大约一半都是循环控制指令，而且函数要把整个矩阵加载并存储 5 次。那么，是否应该把循环展开以减少此种开销呢？若是追求极速算法，若是计算机内存足够用，若是 CPU 指令读取[㊀] 速度快到足以应付大块无循环的"直线型代码" (straight code)，那么可考虑展开。尤其在分支或载入操作很耗时的情况下，更应该如此。展开后的算法程序将反复执行交换位元所用的那 6 条指令，一共要重复 80 遍 (5・16=80)。此外，还要用 32 条加载指令载入源矩阵，并用 32 条存储指令存放它，加总之后，至少是

㊀ 原文为"I-fetching"，似是"Instruction Fetching"（指令读取）的简称。它是"指令流水线"(Instruction pipeline) 上的一个环节。详情参见：https://en.wikipedia.org/wiki/Instruction_pipeline。——译者注

544 条指令。

图 7.8 概述了手工展开之后的程序。此程序并不直接在原矩阵上修改，然而若需要那样做，则它也能根据同一份参数算出正确结果。涉及“交换”操作的代码有 80 行。笔者使用 GNU C 编译器将这段算法编译成可在基本 RISC 架构计算机上执行的程序。算上前后两端的初始（prolog）及收尾（epilog）代码[⊖]，共有 576 条指令（其中没有分支指令，而且不算函数返回所用的指令）。编译用的电脑上没有“多字存储”（store multiple）及“多字载入”（load multiple）指令，不过可以用“双字存储”（store double）及“双字载入”（load double）指令一次存储或载入两个寄存器。

```
#define swap(a0, a1, j, m) t = (a0 ^ (a1 >> j)) & m; \
                           a0 = a0 ^ t; \
                           a1 = a1 ^ (t << j);

void transpose32(unsigned A[32], unsigned B[32]) {
   unsigned m, t;
   unsigned a0, a1, a2, a3, a4, a5, a6, a7,
            a8, a9, a10, a11, a12, a13, a14, a15,
            a16, a17, a18, a19, a20, a21, a22, a23,
            a24, a25, a26, a27, a28, a29, a30, a31;

   a0  = A[ 0];  a1  = A[ 1];  a2  = A[ 2];  a3  = A[ 3];
   a4  = A[ 4];  a5  = A[ 5];  a6  = A[ 6];  a7  = A[ 7];
   ...
   a28 = A[28];  a29 = A[29];  a30 = A[30];  a31 = A[31];

   m = 0x0000FFFF;
   swap(a0,  a16, 16, m)
   swap(a1,  a17, 16, m)
   ...
   swap(a15, a31, 16, m)
   m = 0x00FF00FF;
   swap(a0,  a8,   8, m)
   swap(a1,  a9,   8, m)
   ...
   ...
   swap(a28, a29,  1, m)
   swap(a30, a31,  1, m)

   B[ 0] = a0;  B[ 1] = a1;  B[ 2] = a2;  B[ 3] = a3;
   B[ 4] = a4;  B[ 5] = a5;  B[ 6] = a6;  B[ 7] = a7;
   ...
   B[28] = a28; B[29] = a29; B[30] = a30; B[31] = a31;
}
```

图 7.8　转置 32×32 位矩阵所用的“直线型代码”

要是计算机支持循环移位指令（左右均可），那么有种算法还能再榨取一点性能。此算法的思路是，将图 7.8 所用的交换操作都拿掉，改用一种更简单的方法来交换，新方法无须移位，且每次只需 4 条指令（使用宏来交换，省去移位操作），而旧方法需要 6 条。

⊖ 这两个词借用了文学中的“开场白”和“结束语”，可分别称为“函数序言”和“函数尾声”。在编译器及汇编语言中，前者表示调用函数时准备工作环境所用的指令，而后者指返回时的清理指令。详情参见：https://en.wikipedia.org/wiki/Function_prologue。——译者注

首先，将 $A[16..31]$ 等字组（也就是对所有 $16 \leqslant k \leqslant 31$ 的 $A[k]$）向右循环旋转 16 个位置。然后，将 $A[0]$ 的右半边与 $A[16]$ 的右半边对调，$A[1]$ 的右半边与 $A[17]$ 右半边对调，以此类推，与图 7.8 的代码相似。接下来，将 $A[0..8]$ 等字组与 $A[24..31]$ 等字组分别向右循环移动 8 个位置，然后把 $A[0]$ 与 $A[8]$ 中由掩码 0x00FF00FF 所标注的位元交换，$A[1]$ 与 $A[9]$ 亦然，按照与图 7.8 相似的方式，依次向下执行。上述操作执行 5 轮后，还没有彻底完成转置。最后，要把 $A[1]$ 这个字组循环左移 1 个位置，把 $A[2]$ 循环左移两个位置，依次向下执行（最后这步需要 31 条指令）。本书不列出此算法的代码了，不过会在下面列出 4×4 的位矩阵来演示。

```
abcd       abcd       abij       abij       aeim       aeim
efgh  ==>  efgh  ==>  efmn  ==>  nefm  ==>  nbfj  ==>  bfjn
ijkl       klij       klcd       klcd       kocg       cgko
mnop       opmn       opgh       hopg       hlpd       dhlp
```

图 7.8 中，重排位元的那部分共需 480 条指令（80 次交换操作，每次 6 条）。修改后的程序采用循环移位法，80 次交换各需 4 条指令，加上 80 条循环移位指令（16 • 5 = 80），再加上最后 31 条循环移位指令，共计 431 条。前后两端的初始及收尾指令都没变，所以这样做一共节省了 49 条指令。

另外有种完全不同的位矩阵转置方法，它通过三次“切变换”（shearing transformation）[1]来实现 [GLS1]。若矩阵大小为 $n \times n$，则步骤是：(1) 分别将第 i 行都向右旋转 i 个位置，(2) 分别将第 j 列都向上旋转 $(j+1) \bmod n$ 个位置，(3) 分别将第 i 行都向右旋转 $(i+1) \bmod n$ 个位置，(4) 以矩阵中点为原点建立坐标系，将其元素依横轴做水平镜像。下面演示如何转置 4×4 的位矩阵：

```
abcd       abcd       hlpd       dhlp       aeim
efgh  ==>  hefg  ==>  kocg  ==>  cgko  ==>  bfjn
ijkl       klij       nbfj       bfjn       cgko
mnop       nopm       aeim       aeim       dhlp
```

这个算法并没有其他算法好，因为第 (2) 步的开销太大了。（开销稍微合理一些的方法是，把还需向上旋转的位置大于等于 $n/2$ 的那些列先向上旋转 $n/2$ 个位置 [这些列是第 $n/2-1$ 至第 $n-2$ 列]，然后把某些待旋转的列再向上旋转 $n/4$ 个位置，以此类推。）第 1、3 两步各自只需 $n-1$ 条指令，如果把结果矩阵中的元素都能放在适当位置上，那么第 4 步无需指令。

如果将 8×8 的位矩阵以直观方式存放（把最顶行放在权重最高的 8 个位元处，依次向下摆放），那么矩阵转置操作就相当于三次“外完美洗牌”或“外完美理牌”操作 [GLS1]。若计算机能用一条指令完成“洗牌”或“理牌”操作，则此算法相当好，但是，在基本 RISC 架构的电脑上，它不是个好方法。

---

[1] 作者这里所说的“切变换”，与该操作的经典定义不同。详情参见：https://en.wikipedia.org/wiki/Shear_mapping。——译者注

7.4 压缩算法（广义提取算法）

APL语言[⊖]中有个操作叫“压缩”（compress），其代码写作“B/V”，B是布尔数组，而V是另一个与B同长的数组，其元素任意。该操作的结果是：把B数组中值为“1”的那些位元所对应的V数组元素提取到一个新数组中。存放操作结果的数组，其长度与B数组中值为“1”的元素个数相同。

本书以字组中的位元为例研究类似操作。给定掩码 ***m*** 与字组 ***x***，把 ***x*** 中对应掩码位是“1”的那些位元选出来，并移动（“压缩”）到其右侧。例如，假设待压缩的字组是（每个字母代表一个位元）：

```
abcd efgh ijkl mnop qrst uvwx yzAB CDEF
```

而掩码是

```
0000 1111 0011 0011 1010 1010 0101 0101
```

那么结果就是

```
0000 0000 0000 0000 efgh klop qsuw zBDF
```

仿照许多计算机都支持的“提取”（extract）指令，我们也可将上述操作称为“广义提取”（generalized extract）。

本书想研究的是将此操作在最坏情况下的执行时间降至最短，我们将以图7.9这个简单的循环代码为起点来改进算法。此代码的循环内无分支，在最差情况下，算上函数两端的初始及收尾指令，共需260条。

```
unsigned compress(unsigned x, unsigned m) {
   unsigned r, s, b;    // Result, shift, mask bit.

   r = 0;
   s = 0;
   do {
      b = m & 1;
      r = r | ((x & b) << s);
      s = s + b;
      x = x >> 1;
      m = m >> 1;
   } while (m != 0);
   return r;
}
```

图7.9　用简单的循环实现压缩操作

反复运用平行后缀法（parallel suffix method，参见5.2.1节）执行异或（exclusive

⊖ APL是A Programming Language或Array Processing Language的缩写，由肯尼斯·艾佛森（Kenneth E. Iverson，1920—2004）于20世纪60年代设计。该语言擅长处理数学问题，它以数学符号为基础，其代码用到了一套非ASCII标准的字母。详情参见：https://zh.wikipedia.org/wiki/APL语言。——译者注

or）操作，可改进算法［GLS1］。本书将这种异或法记为 PS-XOR。基本思路是先判明参数 *x* 中有哪些位元需要向右移动奇数个位置，然后移动它们。（先把 *x* 和掩码求“与”运算，屏蔽掉无关位元，这样可简化操作。）掩码也按同样方式移动。接下来，判断 *x* 中有哪些位元需要向右移动 2 倍奇数（例如 2，6，10 等）个位置，然后移动 *x* 中的这些位元，并移动掩码。继续往下，找出 *x* 中需要移动 4 倍奇数个位置的位元，找出需要移动 8 倍奇数个位置的位元，找出需要移动 16 倍奇数个位置的位元。

因为这个据信是源自［GLS1］的算法有点难懂，而且有人也许对这几行算法究竟能做什么还有些疑问，所以本书要详述其操作。假设输入值是：

```
x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF,
m = 1000 1000 1110 0000 0000 1111 0101 0101,
    1    1    111
    9    6    333            4444  3 2  1 0
```

x 中的每个字母表示一个二进制位（值为“0”或“1”）。在掩码 m 中，每个值为“1”的二进制下方都有一个数字[⊖]，表示 x 中的这个位元还需向右移动几个位置。该值也就是 m 中这个位元右侧“0”的个数。前面说过，为计算方便，可先清理 x 中的无关位元，于是：

```
x = a000 e000 ijk0 0000 0000 uvwx 0z0B 0D0F
```

算法首先要找到 x 中哪些位置上的位元要向右移动奇数个位置，然后把它们向右移动 1 位。回忆一下，在 PS-XOR 的操作中，每一个值为“1”的位元都表示原数中此位置及其右侧有奇数个“1”。我们想算的是某一个位置右侧的位元中有多少个“0”。这可通过 mk = ～m≪1 并对 mk 执行 PS-XOR 操作而得到。于是：

```
mk = 1110 1110 0011 1111 1110 0001 0101 0100
mp = 1010 0101 1110 1010 1010 0000 1100 1100
```

可以看出，mk 中值为“1”的位置表示 m 中这个位置紧右侧的那个位元是“0”。而 mp 中的每个位元值都表示把 m 中对应位置右侧“0”的个数加总后模 2。因此，mp 中的“1”也就表示在 m 中这个位置右侧有奇数个“0”。

于是，需要向右侧移动 1 位的位元就是 m 中对应位置右方有奇数个“0”（可通过 mp 中这一位是不是“1”来判断）且原始掩码该位置上是“1”的那些位元。这很容易用 mv = mp & m 来判断：

```
mv = 1000 0000 1110 0000 0000 0000 0100 0100
```

m 中的相关位元可用下面这条赋值语句来移动：

```
m = (m ^ mv) | (mv >> 1);
```

x 中的对应位元可用下面两条赋值语句来移动：

⊖ 上下对齐的“1，3”、“1，6”、“1，9”分别表示 13、16、19。——译者注

```
t = x & mv;
x = (x ^ t) | (t >> 1);
```

（移动 m 比较简单，因为所有待移动的位元都是“1”。）“异或”操作在这里用来把 m 与 x 中的相关位元由“1”变“0”，而“或”的意思则是把 m 与 x 中的相关位元由“0”变“1”。这两个操作也可改用“异或”、“减法”或“加法”。根据由 mv 选出的位置，把 m 与 x 中相关位元都右移 1 位之后，它们变成了：

```
m = 0100 1000 0111 0000 0000 1111 0011 0011
x = 0a00 e000 0ijk 0000 0000 uvwx 00zB 00DF
```

现在需要准备第二次迭代所用的掩码了，它所标注出来的那些位元都还需再向右侧移动 2 的奇数倍个位置才能到位。注意到 mk 中的“1”表示原来 m 中其右侧为“0”的位元，而～mp 中的“1”则表示原来 m 中其右侧“0”的总个数为偶数的位元。这两个论述对修改前的 m 值成立；对第一轮修改后的 m 值来说，如果把前两条论述拆开看，那么每一条都未必成立[㊀]，但若“合起来看”，则必然成立。（换句话说，若把 mk & ～mp 当做新的 mk，则它在修改后的掩码 m 中能够识别出所有其右侧为“0”，且右侧“0”的总个数为偶数的那些位元。）对于新 mk 值的每一个位元来说，如果我们通过 PS-XOR 操作把它右方所有位元都“加总”到其自身，那么把加总后的值当成新 mp，就可以标识出 x 中哪些位元还需要移动 2 的奇数倍（例如 2，6，10 等）个位置。因此，算法确实应该把“mk & ～mp”这个值赋给新 mk，并按前述步骤开始第二轮迭代。修改后的 mk 值为：

```
mk = 0100 1010 0001 0101 0100 0001 0001 0000
```

图 7.10 完整地实现了此操作的 C 语言函数。算上初始及收尾部分，它需要 127 条基本 RISC 指令（该值恒定）[㊁]。图 7.11 中列出了计算过程中几个关键点上的数值序列，其输入值还是上面讨论算法时用的那个。可以看出，该算法还有个副产品，在最后赋完值的 m 中，原来 m 中的“1”全压缩到了右边。

可统计出，图 7.10 的算法在 64 位基本 RISC 架构中，需 169 条指令，而图 7.9 的算法在最坏情况下则需 516 条。

若掩码 m 是常量，则图 7.10 的算法所用指令数还可大幅削减。在遇到如下两种情况时就会如此：（1）“compress (x, m)”这行调用出现在某循环中，m 虽未知，但它的值却在循环内保持不变；（2）m 值已知，compress 函数的代码也许已由编译器预先制成固定指令了。

从图 7.10 中可以看出，循环内对 x 所赋之值并未有其他用途。于是每次的新 x 值仅靠其自身旧值与 mv 即可算出。因此，可把函数代码中涉及 x 的部分删掉，将 5 轮循环所算出的 5 个 mv 值分别存至变量 mv0，mv1，…，mv4 中。这样的话，情况（1）中，原

㊀ 也就是说，在修改后的新 m 中，由旧 mk 中的“1”所标识出的那些位元，其右侧未必就真的是“0”，而由旧～mp 中的“1”所标识出的那些位元，其右侧“0”的个数未必就真是偶数个。——译者注

㊁ 其实第一条左移可以省略，这样就是 126 条了。移位与否均不影响 mv 值［Dalton］。

compress 函数中不涉及 x 的那些代码就可放在调用 “compress (x, m)” 的那个循环外，而把下列代码置于循环内。

```
unsigned compress(unsigned x, unsigned m) {
   unsigned mk, mp, mv, t;
   int i;

   x = x & m;              // Clear irrelevant bits.
   mk = ~m << 1;           // We will count 0's to right.

   for (i = 0; i < 5; i++) {
      mp = mk ^ (mk << 1);             // Parallel suffix.
      mp = mp ^ (mp << 2);
      mp = mp ^ (mp << 4);
      mp = mp ^ (mp << 8);
      mp = mp ^ (mp << 16);
      mv = mp & m;                     // Bits to move.
      m = m ^ mv | (mv >> (1 << i));   // Compress m.
      t = x & mv;
      x = x ^ t | (t >> (1 << i));     // Compress x.
      mk = mk & ~mp;
   }
   return x;
}
```

图 7.10　以“平行后缀法实现压缩”操作

```
             x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF
             m = 1000 1000 1110 0000 0000 1111 0101 0101
             x = a000 e000 ijk0 0000 0000 uvwx 0z0B 0D0F

i = 0,      mk = 1110 1110 0011 1111 1110 0001 0101 0100
After PS,   mp = 1010 0101 1110 1010 1010 0000 1100 1100
            mv = 1000 0000 1110 0000 0000 0000 0100 0100
             m = 0100 1000 0111 0000 0000 1111 0011 0011
             x = 0a00 e000 0ijk 0000 0000 uvwx 00zB 00DF

i = 1,      mk = 0100 1010 0001 0101 0100 0001 0001 0000
After PS,   mp = 1100 0110 0000 1100 1100 0000 1111 0000
            mv = 0100 0000 0000 0000 0000 0000 0011 0000
             m = 0001 1000 0111 0000 0000 1111 0000 1111
             x = 000a e000 0ijk 0000 0000 uvwx 0000 zBDF

i = 2,      mk = 0000 1000 0001 0001 0000 0001 0000 0000
After PS,   mp = 0000 0111 1111 0000 1111 1111 0000 0000
            mv = 0000 0000 0111 0000 0000 1111 0000 0000
             m = 0001 1000 0000 0111 0000 0000 1111 1111
             x = 000a e000 0000 0ijk 0000 0000 uvwx zBDF

i = 3,      mk = 0000 1000 0000 0001 0000 0000 0000 0000
After PS,   mp = 0000 0111 1111 1111 0000 0000 0000 0000
            mv = 0000 0000 0000 0111 0000 0000 0000 0000
             m = 0001 1000 0000 0000 0000 0111 1111 1111
             x = 000a e000 0000 0000 0000 0ijk uvwx zBDF

i = 4,      mk = 0000 1000 0000 0000 0000 0000 0000 0000
After PS,   mp = 1111 1000 0000 0000 0000 0000 0000 0000
            mv = 0001 1000 0000 0000 0000 0000 0000 0000
             m = 0000 0000 0000 0000 0001 1111 1111 1111
             x = 0000 0000 0000 0000 000a eijk uvwx zBDF
```

图 7.11　用“平行后缀法”实现“压缩”运算时的计算过程

```
x = x & m;
t = x & mv0;  x = x ^ t | (t >> 1);
t = x & mv1;  x = x ^ t | (t >> 2);
t = x & mv2;  x = x ^ t | (t >> 4);
t = x & mv3;  x = x ^ t | (t >> 8);
t = x & mv4;  x = x ^ t | (t >> 16);
```

现在循环内只有21条指令（常数加载指令可移至循环外），和完整执行图7.10所需的127条指令相比，改进很大。

若是第二种情况，也就是m值已知，则上述优化依然适用，而且还能继续优化。5个掩码里也许会有0，这时对应的那行代码就可省掉。比如，若m0是0，则说明没有需要移动奇数个位置的位元，若m4是0，则说明没有需要移动超过15个位置的位元，余可类推。

例如：

```
m = 0101 0101 0101 0101 0101 0101 0101 0101
```

则计算出来的掩码是：

```
mv0 = 0100 0100 0100 0100 0100 0100 0100 0100
mv1 = 0011 0000 0011 0000 0011 0000 0011 0000
mv2 = 0000 1111 0000 0000 0000 1111 0000 0000
mv3 = 0000 0000 1111 1111 0000 0000 0000 0000
mv4 = 0000 0000 0000 0000 0000 0000 0000 0000
```

因为最后一个掩码是0，所以如果按照刚才假设的第二种情况，由编译器预先编译好，那么需要17条指令（不计掩码加载）。这不如7.2节那个算法⊖（不计掩码加载，需13条）好。由于那个算法已经预先知道，要从字组中每隔一个位置取一个位元，然后把它们靠右对齐，所以它可利用此特性编出比此更优的代码。

7.4.1 用“插入”、“提取”指令实现压缩操作

若计算机有“插入”（insert）指令，则在刚假设的那种预先编译好的情况下，通常可以用此指令完成“压缩”操作，这种方法需要的指令数比前面讨论的算法少。如果可以把需要存至目标寄存器的位段以立即数表示，并将其作为“插入”指令的操作数，那么效果更好。此外，这种方法也不需要占用寄存器来存放掩码。

先把目标寄存器初始化为0，然后，每当在掩码m中找到一组连续出现的“1”，就把变量x向右移位，使之右对齐到下一个位段，并用“插入”指令将x中的位元放在目标寄存器中适当的位置上。这样做需要$2n+1$条指令，其中n是掩码中的位段个数（也就是有多少组连续出现的“1”）。最坏情况下需要33条指令，因为最多可能出现16个位段（也就是“0”、“1”两种位元值交替出现）。

“插入法”有时能节省很多条指令，例如m=0x0010 084A时。若根据此掩码使用刚

⊖ 作者实际上是指“半理牌”（half unshuffle）形式的“外完美理牌”（outer perfect unshuffle）算法。本例举的m值，恰好就是想在源操作数中每隔一个位置取一个位元，并将其规整到右侧，这与“半理牌”操作一致。——译者注

才讲的压缩方法，则需分别将 x 中的某些位元移动 1、2、4、8、16 个位置。因此，用“平行后缀法”需要执行其全部 21 条指令，而“插入法”只需 11 条（因为掩码中有 5 个位段）。更极端的情况是 m=0x8000 0000。若用“平行后缀法”，则需要把仅有的“1”移动 31 个位置，这需要 21 条指令，但是“插入法”则仅需 3 条，而且，如果不限方法，那么最少只需 1 条指令（也就是右移 31 位）即可。

采用“提取”（extract）指令，也有很多种简单的方法能实现出掩码已知的压缩操作，这需要 $3n-2$ 条指令，其中 n 是掩码中的字段个数。

显然，要想根据已知掩码编译出最佳的压缩操作代码是个困难的问题。

7.4.2　向左压缩

如果想把位元压缩到左侧，显然可以把参数 x 与掩码反转，然后套用向右压缩算法，最后把结果反转。另一种方法是先向右压缩，然后左移 pop($\bar{m}$) 位。若是计算机支持位元反转指令或种群计数指令，那么这些方法都不错，如果不支持，那么简单改编一下图 7.10 的算法就好：除了两个表达式 1≪i 中的移位操作保持不变外，把其余移位操作的方向都反转即可（共需修改 8 处）。

大约于 1967 年问世的 BESM-6 计算机支持“向左压缩”（compress left）函数（“根据掩码 X，把 A 中的位元打包”）及其反操作（“将……解包”），这两个操作都针对计算机中的 48 位寄存器。这些指令不易实现。密码学专家推测，它们的唯一用途就是破译美国人的密码 [Knu8]。BESM-6 也有“种群计数”（population count）指令，前面说过，它似乎对美国国家安全局（National Security Agency）很重要。

7.5　展开算法（广义插入算法）

“向右压缩”函数的反函数将寄存器中的位元由低权重端移到掩码所指定的位置上，位元次序不变。例如，expand（0000 abcd，1001 1010）=a00b c0d0。因此：

$$\text{compress}(\text{expand}(\boldsymbol{x}, \boldsymbol{m}), \boldsymbol{m}) = \boldsymbol{x}$$

此函数又名 unpack（解包）、scatter（分散）、deposit（置入）。

将图 7.10 的代码反向运行即可实现它 [Allen]。为避免覆盖 x 中的原有位元，需要先向左移动那些移动距离较大的位元，最后移动那些仅需移动 1 个位置的位元。这意味着必须先计算并保存前 5 个“移位距离标识符”（也就是代码中的 mv），然后按与计算顺序相反的次序使用之。对很多应用程序来说这不成问题：因为它们要用同一个掩码 m 操作大量数据，所以可预先把这些数算好，然后再按需求复用。

图 7.12 列出了算法代码。它需要大约 168 条基本 RISC 指令（此值恒定），其中包含 5 条存储及 5 条加载指令）。64 位机使用的 64 位版大约需要 200 条指令。

若电脑不支持“and not”指令，那么第二个循环内的“数据选择”（MUX）操作可改写如下，这样能省 1 条指令：

```
x = ((x ^ t) & mv) ^ x;
```

```
unsigned expand(unsigned x, unsigned m) {
   unsigned m0, mk, mp, mv, t;
   unsigned array[5];
   int i;

   m0 = m;              // Save original mask.
   mk = ~m << 1;        // We will count 0's to right.

   for (i = 0; i < 5; i++) {
      mp = mk ^ (mk << 1);              // Parallel suffix.
      mp = mp ^ (mp << 2);
      mp = mp ^ (mp << 4);
      mp = mp ^ (mp << 8);
      mp = mp ^ (mp << 16);
      mv = mp & m;                      // Bits to move.
      array[i] = mv;
      m = (m ^ mv) | (mv >> (1 << i));  // Compress m.
      mk = mk & ~mp;
   }

   for (i = 4; i >= 0; i--) {
      mv = array[i];
      t = x << (1 << i);
      x = (x & ~mv) | (t & mv);
   }
   return x & m0;       // Clear out extraneous bits.
}
```

图 7.12　以“平行后缀法”实现“展开”操作

7.6　压缩与展开操作的硬件算法

本节给出一些实现“向右压缩”及其反操作所用的面向硬件算法（hardware-oriented algorithm）[Zadeck]。与前面几节的算法类似，其执行时间也和计算机字宽的对数成正比。它们适合以硬件方式实现，若用基本 RISC 指令则无法产生高效代码。本书只简述其原理，不列出 C 语言代码或机器码。

7.6.1　压缩

为演示算法操作步骤，我们用字母表示 x 的每个位元，并举一个具体数值当做掩码 m。此二者如下所示：

```
Input x =     abcd efgh ijkl mnop qrst uvwx yzAB CDEF
Mask m =      0111 1110 0110 1100 1010 1111 0011 0010
```

算法要执行 $\log_2(W)$ 个“阶段”（phase），其中 W 是计算机字宽，也就是字组中的位元数。每个阶段都要并行操作数个大小为 2^n 位的“小块”（pocket），n 的取值范围从 1 到 $\log_2(W)$。每个阶段执行完毕后，x 中每“小块”中的位元已按照掩码 m 的对应“小块”，从原来 x 的该“小块”中选出并靠右压缩好了。此时 m 的每个“小块”都含有一个整数，它表示 m 中该“小块”原来有多少个值为“0”的位元。这也是原来 x 中没有被选中并压缩到右侧的位元数。这些位元是现在 x 中每“小块”中的前导“0”。

每一阶段，算法都在 $\boldsymbol{x}$ 与 $\boldsymbol{m}$ 的每个“小块”中并发执行下列步骤，其中 w 表示每“小块”中的位元个数。

1. 设定值 $\boldsymbol{L}$，令其左半侧等于 $\boldsymbol{x}$ 中某“小块”的左半侧，其右半侧为 $w/2$ 个“0”。

2. 右移 $\boldsymbol{L}$ 的全部 w 个位元，移位量是 $\boldsymbol{m}$ 中对应“小块”的右半侧，移位时左方空出来的位元填“0”。该操作不可能把“1”移出去，因为最大移位量就是 $w/2$。

3. 设定值 $\boldsymbol{R}$，令其左半侧为 $w/2$ 个“0”，其右半侧是 $\boldsymbol{x}$ 中对应“小块”的右半侧。

4. 把 $\boldsymbol{R}$ 与移位后的 $\boldsymbol{L}$ 取或，用此结果替换 $\boldsymbol{x}$ 中该“小块”的全部 w 个位元。

5. 将 $\boldsymbol{m}$ 的左右两半加总，用求和结果替换 $\boldsymbol{m}$ 中该“小块”的全部位元。

为了执行第一阶段（也就是 $w=2$ 时），需要先对 $\boldsymbol{x}$ 和 $\boldsymbol{m}$ 求“与”，以清除 $\boldsymbol{x}$ 中的无关位元，然后还要把 $\boldsymbol{m}$ 取补，这样的话，新 $\boldsymbol{m}$ 中的每一位都表示：在这长度为 1 的半个“小块”中有多少个“0”。把第一阶段当成特例来处理会简单些：先执行刚才说的两个操作，然后根据下表所列逻辑来计算 $\boldsymbol{x}$ 和 $\boldsymbol{m}$ 中每个 2 位“小块”，以便执行第一轮压缩操作。

输入		输出	
x	m	x	m
ab	00	00	10
ab	01	0b	01
ab	10	0a	01
ab	11	ab	00

例如，表中第 3 行的 $\boldsymbol{m}=10$（二进制）。这意味着将 $\boldsymbol{x}$ 的左位元选入结果，而右位元没入选。因此，左位元（也就是 a）被压缩至右侧。将 $\boldsymbol{x}$ 的另一个位元清零，这样可确保在最终结果中高权重位（表示未入选的位元）都是 0。

对初始的 $\boldsymbol{x}$ 与 $\boldsymbol{m}$ 运用上述逻辑，可得：

```
位元对   x = 0bcd ef0g 0j0k mn00 0q0s uvwx 00AB 000E
         m = 0100 0001 0101 0010 0101 0000 1000 1001
```

我们以上述 $\boldsymbol{x}$ 中的第二个半字节（ef0g）来演示第二阶段。先构建 $\boldsymbol{L}$＝ef00，$\boldsymbol{R}$＝000g。将 $\boldsymbol{L}$ 右移 1 位（移位量等于 $\boldsymbol{m}$ 中对应半字节的右半侧），变成 0ef0。将此值与 $\boldsymbol{R}$ 取“或”，得出此半字节的新值：0efg。将 $\boldsymbol{m}$ 的左右两半相加，新的 $\boldsymbol{m}$ 值是 0001（新值和旧值一样）。

```
半字节   x = 0bcd 0efg 00jk 00mn 00qs uvwx 00AB 000E
         m = 0001 0001 0010 0010 0010 0000 0010 0011
```

同理，第三、第四、第五阶段会分别压缩 $\boldsymbol{x}$ 的每个字节、半字、字组，并更新 $\boldsymbol{m}$，演示如下：

```
字节   x = 00bc defg 0000 jkmn 00qs uvwx 0000 0ABE
       m = 0000 0010 0000 0100 0000 0010 0000 0101
半字   x = 0000 00bc defg jkmn 0000 000q suvw xABE
       m = 0000 0000 0000 0110 0000 0000 0000 0111
```

```
字      x = 0000 0000 0000 0bcd efgj kmnq suvw xABE
        m = 0000 0000 0000 0000 0000 0000 0000 1101
```

运算完毕后，**m** 这个整数就表示现在 **x** 中前导 0 的个数。用字宽减去该值，就是当前压缩在 **x** 中的位元数，也就等于原始掩码 **m** 中“1”的个数。

如果用基本 RISC 指令集来实现，那么这个算法不太好，因为难于找到能同时将每个“半小块”（half-pocket）向右移动不同长度的办法。另一方面，在“SIMD 计算机”[⊖]上此算法也许有用，因为它们有指令可以并发操作字组中各个“小块”而不互相干扰。

7.6.2 展开

实际上只要把“硬件压缩算法”先正向执行一次，再逆向执行一次，即可实现展开算法。与基于“平行后缀法”的算法一样，该算法也要把“硬件压缩算法”里的五个掩码计算并保存起来，然后按照与计算次序相反之顺序使用它们。其实最后一个掩码用不上（在压缩算法里也没用到），不过，要多计算一个掩码 **m0**，它是原掩码的反码。在正序执行时，只需计算掩码，涉及数据 **x** 的步骤可省略。

为了演示此算法，假定：

```
Input x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF
Mask  m = 0111 1110 0110 1100 1010 1111 0011 0010
```

“展开”操作的结果应该是：

```
0nop qrs0 0tu0 vw00 x0y0 zABC 00DE 00F0
```

算法所用掩码如下：

```
m0 = 1000 0001 1001 0011 0101 0000 1100 1101
m1 = 0100 0001 0101 0010 0101 0000 1000 1001
m2 = 0001 0001 0010 0010 0010 0000 0010 0011
m3 = 0000 0010 0000 0100 0000 0010 0000 0101
m4 = 0000 0000 0000 0110 0000 0000 0000 0111
```

m4 左右两半的整数值表示原始掩码 **m** 的左右两半分别有多少个“0”。尤其注意 **m** 右半边有 7 个“0”。这意味着原 **x** 右半边权重较高的 7 个位元在最终结果里不会出现于右半边，它们应该会在左半边。所以说，应该把 **x** 的 9 至 15 号位元左移足够的距离，以便将其放在 **x** 左半边，而为了适应此操作，原 **x** 里权重较高的位元也要往左移。要想做到这一点，可把 **x** 全部 32 位都左移 7 个位置，用移位之后的左半边取代现有 **x** 的左半边。于是可得：

```
x = hijk lmno pqrs tuvw qrst uvwx yzAB CDEF
```

一般来说，算法要经由掩码 **m4** 至 **m0**，分 5 个阶段把“小块”的长度由 32 缩减至 2。每个阶段都要同时左移各“小块”，将移出“小块”左边界的位元丢弃，于右侧空位补

⊖ Single Instruction Multiple Data，单指令流多数据流。SIMD 是以费林分类法（Flynn's Taxonomy）所定义的计算机类型。它能同时对一组数据中的每个元素分别执行同一种指令。详情参见：https://zh.wikipedia.org/wiki/单指令流多数据流。——译者注

零，以便使移位之后的“小块”与移位之前等长。然后用移位后的左半边替换掉原“小块”的左半边。这会在“小块”左右两半都留下“垃圾”位元。执行完最后一个阶段后，会把结果同原始掩码求“与”运算，以将那些位元清零。

继续向下执行，将 ***m*3** 看做两个 16 位“小块”。左“小块”的右半边相当于十进制整数 4，所以把 ***x*** 的左“小块”左移 4 位（于是得到 `lmno pqrs tuvw 0000`），然后用这个值的左半边替换该“小块”的左半边，现在 ***x*** 的左“小块”是 `lmno pqrs pqrs tuvw`。对 ***x*** 右“小块”里的 16 个位元，也执行同样操作，于是可得：

```
x = lmno pqrs pqrs tuvw vwxy zABC yzAB CDEF
```

下个阶段使用 ***m*2** 来操作长度为 8 位元的各“小块”。执行完该阶段后，***x*** 是

```
x = mnop pqrs rstu tuvw vwxy zABC BCDE CDEF
```

再下个阶段使用 ***m*1** 来操作长度为 4 位元的各“小块”。执行完该阶段后，***x*** 是

```
x = mnop qrrs sttu vwvw wxxy zABC BCDE DEEF
```

末尾阶段使用 ***m*0** 来操作长度为 2 位元的各“小块”。执行完该阶段后，***x*** 是

```
x = mnop qrss stuu vwww xxyy zABC CCDE EEFF
```

最后一步是将上述 ***x*** 和原始掩码求“与”，以清除无关位元。于是得出

```
x = 0nop qrs0 0tu0 vw00 x0y0 zABC 00DE 00F0
```

掩码 ***m*0** 至 ***m*4** 中，每半“小块”内的数字就等于原始掩码 ***m*** 在这半“小块”内“0”的个数。因此，除了像刚才那样计算并存储掩码外，还有个办法，就是利用“种群计数”电路“即时”（on the fly）算出 ***m*** 中每半“小块”内“0”的个数。

7.7 通用置换算法及分羊操作

无论是想重新排列字组中的位元，还是想重排其他事物，所面临的一个核心问题就是如何表达这个重排动作。没有很简洁的表现形式，因为要重排 32 位字组中的位元共有 32！种方法，这需要用至少⌈log2（32!）⌉=118 个位元来表示，也就是需要 3 个字组外加 22 个位元才能指明这 32！种排列方式中的某一种。

有一种有趣的形式可以表达重排动作，它与 7.4 节说的压缩操作密切相关［GLS1］。首先把每个位元要移动到的位置直接列出来。比方说，在“循环左移 4 位”这个重排操作中，0 号位元（最低有效位）移动到了 4 号位置，1 号位元移动到了 5 号位置，……，31 号位元移动到了 3 号位置。此重排操作可用包含 32 个 5 位元索引的数组表示：

```
00100
00101
...
11111
```

```
00000
00001
00010
00011
```

把上述内容看成位矩阵，我们考虑其转置矩阵，不过这次转置要沿着“次对角线”(off diagonal)[㊀]翻转，以便转置之后的首行包含上述32个索引的最低有效位（假设计算机使用“小端序”(little endian)）。把转置后的5个32位字组存入数组p：

```
p[0] = 1010 1010 1010 1010 1010 1010 1010 1010
p[1] = 1100 1100 1100 1100 1100 1100 1100 1100
p[2] = 0000 1111 0000 1111 0000 1111 0000 1111
p[3] = 0000 1111 1111 0000 0000 1111 1111 0000
p[4] = 0000 1111 1111 1111 1111 0000 0000 0000
```

x中某位置上的位元，其最终位置的最低有效位保存在p[0]的对应位元中。次低有效位保存在p[1]的对应位元中，依此类推。这种编码方式与前几节说的掩码mv类似，不过mv是运用于压缩算法中修改后的掩码上，而非运用于原始掩码。

现在需要执行的压缩操作是把x中对应掩码位是“1”的所有位元压缩至左端，对应掩码位是“0”的所有位元压缩至右端[㊁]。这有时也叫“绵羊与山羊分离操作”（分羊法，sheep and goats operation，SAG）[㊂]或“广义理牌”操作。它可用下式计算：

```
SAG(x, m) = compress_left(x, m) | compress(x, ~m).
```

将SAG视为一种基本操作，用上面说的数组p表示重排动作，这样的话，用下列15个步骤即可把字组x中的位元按数组p所示方法重排：

```
x    = SAG(x,    p[0]);
p[1] = SAG(p[1], p[0]);
p[2] = SAG(p[2], p[0]);
p[3] = SAG(p[3], p[0]);
p[4] = SAG(p[4], p[0]);

x    = SAG(x,    p[1]);
p[2] = SAG(p[2], p[1]);
p[3] = SAG(p[3], p[1]);
p[4] = SAG(p[4], p[1]);

x    = SAG(x,    p[2]);
p[3] = SAG(p[3], p[2]);
p[4] = SAG(p[4], p[2]);

x    = SAG(x,    p[3]);
p[4] = SAG(p[4], p[3]);

x    = SAG(x,    p[4]);
```

㊀ 也就是从右上至左下的那条。与之相对，一般所说的转置操作，要沿着从左上到右下的那条主对角线翻转。——译者注

㊁ 如果计算机用“大端序”(big-endian)来编码位元，那么就把掩码位是“0”的对应x位元压缩至左端，是“1”的压缩至右端。

㊂ 此操作借用了《圣经·马太福音》里的典故。耶稣把信徒分为两队，能达到“彼此相爱”境界的是可得救的绵羊，不能者为不可得救的山羊。后来用绵羊与山羊分别比喻善人与恶人。详情参阅：http：//zh.wikipedia.org/wiki/分羊的比喻。——译者注

在上述步骤中，SAG 用于执行“稳定二进制基数排序”（stable binary radix sort）[⊖]。数组 p 保存了 32 个 5 位元的键，它们用于排列 x 中的位元。第一步是把 x 中对应 p［0］位置上为“1”的那些位元移至左侧，把其他那些对应 p［0］位置上为“0”的位元移至右侧。执行此操作时，同值位元之间的次序在 SAG 操作之后没变（这也就是说，此排序操作是“稳定的”）。然后，所有要参与下轮排序的键，其本身也需按类似方式排序。第 6 行代码的意思是根据键的“次低有效位”（second least significant）对 x 排序。

与执行压缩操作时的情况类似，若以 p 表示的排列动作要运用于多个 x 字组，那么可以预先计算上述步骤，这样能大幅节省时间。表示排列动作的数组 p 可改用如下方式计算：

```
p[1] = SAG(p[1], p[0]);
p[2] = SAG(SAG(p[2], p[0]), p[1]);
p[3] = SAG(SAG(SAG(p[3], p[0]), p[1]), p[2]);
p[4] = SAG(SAG(SAG(SAG(p[4], p[0]), p[1]), p[2]), p[3]);
```

这样的话，每一种排列操作，都可用下列代码完成：

```
x = SAG(x, p[0]);
x = SAG(x, p[1]);
x = SAG(x, p[2]);
x = SAG(x, p[3]);
x = SAG(x, p[4]);
```

有种更直接（但可能不太有趣）的方法能实现通用排列，那就是把针对字组 x 中每个位元的排列动作表示为 32 个连续出现的 5 位元索引。第 k 个索引值表示操作结果中的第 k 个位元相当于源操作数的第几个位元。（这是个表示数据来源的列表（“comes from” list），而不是 SAG 操作所用的那种表示数据去向的列表（“goes to” list）。）这些索引可以打包成 6 个 32 位字组，也就是需要 6 个字组来容纳全部 32 个位索引。可用硬件方式实现如下指令：

```
bitgather Rt,Rx,Ri,
```

其中 Rt 寄存器是目标寄存器（同时也是源寄存器），Rx 寄存器中含有待重排的位元，而 Ri 寄存器中装有 6 个 5 位元索引（还有两个位元没用到）。指令要执行的操作是：

$$t \leftarrow (t \ll 6) \mid x_{i_0} x_{i_1} x_{i_2} x_{i_3} x_{i_4} x_{i_5}$$

目标寄存器里的内容要左移 6 位，然后从 x 中选出 6 个位元填补 t 中空出来的 6 个位置。入选位元由字组 i 中由左至右的 6 个 5 位元索引决定。索引中的位元编码顺序可能是“小端序”或“大端序”，而此操作在这两种电脑中也许都有对应实现方式。

要重排字组，需连续使用 6 次上述指令，每次用的 Rt 与 Rx 都和上次相同，但索引寄

⊖ 基数排序（radix sort）是一种不通过直接比较整个元素值而实现的整数排序算法，原理是将整数按其每个位数分别比较。将所有待比较的正整数统一为相同数位长度，较短的前面补零。从最低位开始排序，然后次低位，直至最高位，排完之后的数列就有序了。详情参见：https://zh.wikipedia.org/wiki/基数排序。——译者注

存器不同。首个索引寄存器中，只有 i_4 与 i_5 两索引有用，因为由其余 4 个索引选出来的位元会移出 Rt 的左边界。

此指令在实现时很可能允许使用重复的索引，所以它除了重排位元顺序外还有其他功能。它可以把由目标寄存器里随意选出来的位元重复任意次数。而 SAG 操作则不这么通用。

想把这项操作实现成一条快速指令（比如只需 1 个周期）也不太难。“位选择电路”（bit selection circuit）是由 6 个“32 选 1 的数据选择器”（32：1 MUX）组成。如果按照当前科技水平，用“2 选 1 的数据选择器”（2：1 MUX）分 5 个阶段实现（共需 6 · 31=186 个数据选择器），那么此指令要比 32 位加法指令快 [MD]。

某些 Intel CPU 也有一些指令，其功能和上述位元排列操作很像，不过它们排列的元素是一系列字节、“字组”（这种字组是 16 位）、“双字”（这种双字是 32 位）。这些指令分别叫做 PSHUFB、PSHUFW、PSHUFD（Shuffle Packed Bytes/Words/Doublewords）[一]。

位元排列操作可用于密码学，也同计算机图形学中的“子字组”排列（例如，排列字组里的字节）密切相关。这两种应用场景更有可能处理 64 位或 128 位字组，而非 32 位字组。显然，只要修改一下 SAG 与 bitgather 方法，就能适应这些字组长度更大的情况了。

为了用 DES（Data Encryption Standard，数据加密标准）[二]算法加密或解密一条消息，需要大量执行一种与重排操作类似的映射操作（permutation-like mapping）。首先，要为每次会话生成一个密钥。这涉及 17 次与重排类似的映射操作。其中的首个操作名为“选择置换 1”（Permuted Choice 1），它把 64 位值映射成 56 位值（从密钥中选出 56 个“非奇偶校验位”（non-parity bit）并将其重排）。然后，将相同的映射操作执行 16 次，每次操作都叫做“选择置换 2”（Permuated Choice 2）[三]，该操作会把 56 位值映射为 48 位值。

生成密钥后，要在信息中的每个 64 位块上执行 34 次类似重排的映射操作。第一次及最后一次都是 64 位排列操作，且两者互为逆运算。还有 16 次重复的置换操作，它们都使用相同的映射规则，将 32 位量映射成 48 位量。除此而外，又有 16 次 32 位重排操作，这 16 次使用的重排规则也相同。于是总共有 6 种不同的映射操作，它们都根据由常量所定义的固定映射表来执行，其内容可参阅 [DES]。

DES 算法已经废弃了，因为在 1998 年“电子前锋基金会”（Electronic Frontier Foundation）[四]用特殊硬件证实了它并不安全。美国国家标准技术研究所（National

---

㊀ 这些指令的详情可参考：http：//download.intel.com/products/processor/manual/325462.pdf。——译者注

㊁ 该标准的详情可参见：https：//zh.wikipedia.org/wiki/数据加密标准。——译者注

㊂ 这两种选择置换操作分别简称为“PC-1”及“PC-2”，它们与后文提到的另外 4 种重排操作均可参见：https：//zh.wikipedia.org/wiki/DES 补充材料。——译者注

㊃ 或译“电子前哨基金会”、“电子前线基金会”、“电子前沿基金会”，是国际知名的法律援助公益组织，旨在宣传互联网版权和监督执法机构，总部设在美国。详情参见：https：//zh.wikipedia.org/wiki/电子前锋基金会。——译者注

Institute of Standards and Technology，NIST）同意使用一种名为“Triple DES”[⊖]的临时方案代替它，该方案需要在每个 64 位块上连续执行 3 次 DES 算法，每次使用不同的密钥（也就是说，密钥为 192 位，其中含 24 个奇偶校验位）。因此，加密或解密时所需执行的重排操作是 DES 算法的 3 倍。

DES 及 Triple DES 加密法的“永久”替换方案是“高级加密标准”（Advanced Encryption Standard，AES，原来叫做“Rijndael 算法”［AES］），它不需要执行位元级别的重排（bit-level permutation）。与其最接近的操作就是以 8 的倍数为移位量简单地旋转 32 位字组而已。其他已提议或投入使用的加密算法，其位元级别的重排操作数都比 DES 少。

现在比较一下本节所讲的两种重排算法。bitgather 方法的好处有：（1）能够较为简单地从原始数据中安排好描述重排操作的索引字组；（2）用硬件实现起来更容易；（3）能执行更加通用的映射方案。SAG 方法的好处有：（1）只需 5 条指令即可，而不像 bitgather 那样需要 6 条；（2）指令格式中只用了两个源寄存器（这可能令其更适合在某些 RISC 架构中实现）；（3）可较好地扩展成适用于双字的重排算法；（4）能更高效地重排“子字组”。

［LSY］一文讨论了第 3 条。想要用 SAG 指令实现双字组的通用排列操作，可以执行两次 SAG 指令、一些基本 RISC 指令，以及两次单字组“全排列”（full permutation）操作。如用 bitgather 指令来做，则需要执行三次单字组全排列，外加一些 RISC 指令。预先处理描述排列操作的值，用以生成只依赖其值的新值——这些操作所耗指令并未算在刚才说的那些之内。这两种算法具体如何扩展，留待读者探索。

现在谈谈第 4 条。比如说，想要用 bitgather 重排字组中的 4 个字节，那么需要 6 条指令，这个指令数与用 bitgather 执行通用置换时一样。但若用 SAG，则只需两条指令，而不像用它执行通用位元置换算法时那样需要 5 条。就算“子字组”长度不是 2 的幂，其效率也照样胜过前者。它需要$\lceil \log_2 n \rceil$步来执行“子字组”长度为 n 的位元替换操作，这不包含可能出现在两端但却不参与排列的位组。

［LSY］一文讨论了 SAG 和 bitgather 指令（文中分别称为“GRP”与“PPERM”），也讨论了其他一些可行且基于网络的置换指令，以及查表置换法。

有个巧妙的方法能向“山羊”群里多放一只“山羊”，也就是说，它可以计算出

$$\mathrm{SAG}^{-1}(\mathrm{SAG}(\boldsymbol{x},\boldsymbol{m})+1,\boldsymbol{m})$$

的值，但却不使用 SAG 函数及其逆运算［Knu8］。此处假定 SAG($\boldsymbol{x}$，$\boldsymbol{m}$）把“山羊”放在右边，而且加法操作不会溢出到“绵羊”那边去。探寻此技巧的乐趣，就留给读者来享受吧。

7.8　重排与下标变换

许多简单的计算机字组位元重排操作均可分别用更简单的坐标变换、下标变换或位元变换来实现［GLS1］。只要数组元素个数是 2 的整数幂，那么这些变换便可用于重排任

⊖ 也称 3DS 或“三重数据加密算法”，详情参见：http：//zh. wikipedia. org/wiki/3DES。——译者注

意一维数组中的元素。在程序设计领域，这些技巧主要用在其元素与计算机字组等长或更大的数组上。

以“外完美洗牌”操作为例，数组 A 中的 8 个元素要按照如下方式分别移动到其在结果数组 B 中的正确位置上：

$$A_0 \to B_0;\quad A_1 \to B_2;\quad A_2 \to B_4;\quad A_3 \to B_6;$$
$$A_4 \to B_1;\quad A_5 \to B_3;\quad A_6 \to B_5;\quad A_7 \to B_7;$$

B 的每一个下标都等于用三位元旋转器（3-bit rotator）把 A 的对应下标向左旋转 1 位后的结果。显然，把 A 的每个下标向右旋转 1 位，就可实现“外完美理牌”操作。表 7.1 列出了其他几组对应操作。n 是数组元素个数，“lsb”表示最低有效位，下标旋转操作用 $\log_2 n$ 位的旋转器执行。

表 7.1　相互对应的重排与下标变换操作

重排	下标变换	
	数组下标，或“大端序”的二进制数	“小端序”的二进制数
反转位元	求反码	求反码
位元翻转，或者叫广义的位元反转	与常数求“异或”	与常数求“异或”
循环左移 k 位	减 k（模 n）	加 k（模 n）
循环右移 k 位	加 k（模 n）	减 k（模 n）
外完美洗牌	循环左移 1 位	循环右移 1 位
外完美理牌	循环右移 1 位	循环左移 1 位
内完美洗牌	循环左移 1 位，然后将最低有效位取反	对最低有效位取反，然后循环右移 1 位
内完美理牌	对最低有效位取反，然后循环右移 1 位	循环左移 1 位，然后将最低有效位取反
转置存放与 64 位字组中的 8×8 位矩阵	向左或向右旋转 3 位	向左或向右旋转 3 位
快速傅里叶变换中的“还原”步骤（FFT unscramble）	反转位元	反转位元

7.9 LRU 算法

有没有想过计算机如何知道哪个缓存段（cache line）的上一次使用时间距离现在最久？本书描述一种称为“引用矩阵法”（reference matrix method）的算法。它主要以硬件算法形式实现，不过也许在软件编程中也有用。

本书不打算花大篇幅讲述有趣的缓存问题，不过读者要知道，计算机的主存（main memory）与处理器（processor）之间有缓冲数据用的高速缓存[⊖]。缓存也许在每个计算

⊖ 这些高速缓存块就是作者刚才提到的“缓存段”，详情参见：http：//en.wikipedia.org/wiki/CPU_cache#Cache_entries。——译者注

机周期中都会收到获取某字组的请求，而它们通常要在一两个周期内返回数据，所以没时间执行太复杂的算法。

缓存拷贝了主存中的一个数据子集，而我们要解决的问题是，如果发生“缓存未命中”(cache miss)[1]现象（也就是有程序向缓存请求某地址中的字组，而缓存里却没有该地址中的数据），那么计算机为了缓存新请求的数据，它应该丢弃哪一块现有缓存（或者用缓存术语来说，哪一个“缓存段”）呢？理想的做法应该是，把下一次使用时间距离现在最久的那块数据替换掉。但是我们无法预知未来，所以只能猜测了。对各种各样的应用程序来说，最合理的猜测方法似乎是 LRU 策略（least recently used，上一次使用时间距离现在最久）。此策略会将上次使用时间距现在最久的那块缓存替换掉。

缓存有三种：直接匹配式（direct-mapping）、全结合式（full associative）、组结合式(set-associative)[2]。在直接匹配式缓存中，对于加载指令或存储指令所用到的地址来说，计算机会根据地址值中的某些位元，直接把此地址映射到特定的缓存段上。如果未命中缓存，那么毫无疑问，应该把和指令中的地址相对应的那一块替换掉。此时不需要运用 LRU 或其他猜测法。

全结合式缓存中，主存里的一块数据可以放在任意缓存块中。执行加载或存储指令时，首先看其用到的地址是否在缓存中。如果没在，那么就必须把某个缓存块的内容替换掉。计算机可以自由选择需替换掉的缓存块。可用的算法很多（其中常用的是：FIFO(First in first out，先进先出)、随机、LRU)，不过正如刚才所说，似乎很多时候 LRU 算法都能带来最低的未命中率“miss rate”。不巧的是，如果可供替换的缓存块有很多，那么 LRU 会是实现成本最高的一种决策法。

通常会以组结合的形式组织缓存。它是直接匹配式与全结合式的折中。设计者决定其结合程度，通常是 2、4、8、16。缓存分为若干“组”(set)，每一组通常有 2、4、8 或 16 个缓存块（line）。对于加载或存储指令中用到的地址来说，可以直接根据地址值中的某些位元定位到具体的“组”，不过组里面的块就必须要查找了。在组中寻找块的方式与全结合式缓存所用方式非常相似。现在假设必须替换掉其中某一块。LRU 算法只需要在给定的组内判断哪一块缓存上次的使用时间距离现在最久，并将其替换即可。

有了这些简要的背景知识后，就可以描述“引用矩阵法”了。为了演示此算法，我们假定使用 4 路组结合式（four-way set associative）缓存。缓存中的每个关联组都配有这样一个 4×4 矩阵。算法的核心是，当第 i 个缓存块被引用后，把矩阵里第 i 行元素设为 1，然后将第 i 列元素设为 0。图 7.13 依次演示了在初始状态及第 3、1、0、2、0、3、2 块缓存被引用时引用矩阵的样子。

在每个矩阵的四行中，有一行含有三个“1”，有一行含有两个“1”，有一行含有一

---

[1] 也可称“缓存落空”。——译者注

[2] 其中“全结合式”也称“全相连式”、“全关联式”；“组结合式”也称“组相连式”、“组关联式”、“集合关联式”。——译者注

缓存块	初始状态 0123	3 0123	1 0123	0 0123	2 0123	0 0123	3 0123	2 0123
0	0111	0110	0010	0111	0101	0111	0110	0100
1	0011	0010	1011	0011	0001	0001	0000	0000
2	0001	0000	0000	0000	1101	0101	0100	1101
3	0000	1110	1010	0010	0000	0000	1110	1100

图 7.13 演示“引用矩阵法”

个“1”，还有一行没有“1”。没有“1”的那行，其行号也就代表上次使用时间距现在最远的那块缓存，“1”的个数越多，表明上次使用时间据现在越近。当发生“缓存落空”时，计算机找到全是“0”的那一行，并且将相关缓存替换掉。其后，它把该行元素全设为 1，表示此缓存块是刚使用过的，然后将该列元素置 0。

为什么可以用这个矩阵实现 LRU 算法呢？因为根据矩阵 M 的元素 M_{ij} 就可以看出第 i 块缓存上次使用时间是不是比第 j 块距离现在更近。若 $M_{ij}=1$，那么第 i 块缓存使用时间就比第 j 块距现在更近，若 $M_{ij}=0$，则表明前者不比后者近。

假设有个 4×4 矩阵，其初始值不限，现在，第 2 块缓存被引用了。那么，矩阵就要按图 7.14 所示改变其元素值。除了在主对角线（main diagonal）上的那个元素外，将第 i 行其他元素设为 1，以表明第 i 块内存上次使用时间比第 j 块近（j 为不等于 i 的其他值）。将第 i 列元素置 0，以表明第 j 块缓存上次使用时间并不比第 i 块更近（对所有的 j 来说都是如此）。对于序号都不等于 i 的两个缓存块来说，其上次使用时间的先后关系没有改变。只要每个缓存块都引用一次，那么整个“使用时间先后关系表”就建立起来了。

缓存块	初始状态 0123	2 0123
0	abcd	ab0d
1	efgh	ef0h
2	ijkl	1101
3	mnop	mn0p

图 7.14 “引用矩阵法”中的一个步骤

因此，引用矩阵中所表达的关系是“抗对称的”（antisymmetric）⊖，并且其主对角线上的元素总为 0。于是，只要在缓存中保存主对角线右上方或左下方的元素就可以了。在实践中也是这么做的。对于 n 路组结合式缓存来说，需要用 $n(n-1)/2$ 个位元来存储这个引用矩阵。如果 $n=4$，那么就是 6 个，如果 $n=8$，则是 28 个。28 已经显得有点大了，所以“引用矩阵法”以及真实场景中的 LRU 策略都不太会用在结合度大于 8 的缓存上。在那种情况下，可以使用类似 LRU 的方法，或者根本不是 LRU 的其他方法。

如果用软件实现，那么也许会把每个缓存块的序号保存在列表（用一个简单的数组或链表来表示）里。如果第 i 个缓存块被引用了，那么就在列表里找到 i，并把 i 移到列表顶端。这样的话，列表底部的那个数字就表示其上次使用时间距离现在最远的那块缓存。

⊖ 也叫“反对称的”，它与“对称”（symmetric）的反义词“不对称”（asymmetric）含义不同。对于集合 X 中两个不相等的 a、b 值来说，如果二元关系 $R(a, b)$ 成立意味着 $R(b, a)$ 必然不成立，则 R 就是“反对称的”。换句话说，对于某个“反对称的 R”来说，若集合 X 中某对 a、b 值能使 $R(a, b)$ 与 $R(b, a)$ 同时成立，则 a、b 必然相等。详情参见：http://en.wikipedia.org/wiki/Antisymmetric_relation。本例中的 X 为下标取值范围集合 {0, 1, 2, 3}，a、b 为矩阵行坐标或列坐标，$R(a, b)$ 可表述为：由行坐标 a 及列坐标 b 所确定的矩阵元素值为 0。——译者注

上述方法在发生引用时执行得相对较慢（因为要重新排列表中的元素），而在决定应该替换哪块缓存时则执行得较快。还有一种方法在这两种情况下的执行速度相反。这个方法是，拿一个长度与结合度相等的数组，把序号为 i 的缓存块所对应的内存地址存在位置 i 上，同时还把这块缓存的“age”（年代，其实表示它的“新鲜程度”（newness））也保存在这个位置上。如果缓存块 i 被引用了，那么把存放当前“age”的那个变量递增 1，然后将此值存放在数组的第 i 个位置上。在数组中搜索“age”值最低的那个位置，就能找到上次使用时间距现在最远的缓存块了。若保存“age”的整数变量溢出，则此方法失效。

也许每块组结合式缓存中都有一个“age”整数变量，或是全部缓存里只有一个这种变量；在硬件 LRU 算法中，也可以用“周期计数器”（cycle counter）[⊖] 实现此功能。

当结合度较小时，“引用矩阵法”对软件 LRU 算法来说也许有用。例如，假设某个程序使用 64 位计算机上的 8 路组结合式缓存。那么，只用一个 64 位寄存器即可存放“引用矩阵”。用寄存器里权重最低的 8 位保存矩阵第 0 行数据，接下来 8 位保存第一行，以此类推。当组里的第 i 块缓存被引用时，把寄存器中字节 i 的每个位元都设为“1”，然后把序号为 i，$i+8$，…，$i+56$ 的位元清零。用 $\boldsymbol{m}$ 表示寄存器，可将该操作描述如下。

$$\boldsymbol{m} \leftarrow \boldsymbol{m} \mid (\mathbf{0xFF} \ll (8 * \boldsymbol{i}))$$
$$\boldsymbol{m} \leftarrow \boldsymbol{m} \,\&\, \neg(\mathbf{0x01010101\ 01010101} \ll \boldsymbol{i})$$

上两式共需 5 或 6 条指令，外加一些常量加载指令。只要搜索其位元值全是“0”的字节，就能找出上次使用时间距离现在最远的缓存块了（参见 6.1 节）。与刚才简述的那些软件实现法相比，这个方法的好处是，所有操作都能在寄存器内完成。

7.10 习题

1. 解释第二个 Möbius 公式的原理（7.1.3 节最后的等式（1））。
2. “外完美洗牌”操作与其逆操作使用下列掩码：

m_0 = 0x22222222
m_1 = 0x0C0C0C0C
m_2 = 0x00F000F0
m_3 = 0x0000FF00

   那么，生成 m_k 的通用公式是什么？如果待混排的整数其长度上限无法预知，那么这样的公式也许有用，比如在那种使用高精度计算的应用程序里。
3. 编一个与图 7.9 相似但能执行“展开”操作的函数。
4. 对于 n 路组结合式缓存来说，理论上只需多少个二进制位即可实现 LRU 算法？挑几个比较小的 n 值，对比上述方法和“引用矩阵法”分别需要的位元个数。

⊖ 在某些架构中也叫“时间戳计数器”（Timer Stamp Counter，TSC），是 CPU 里统计周期的高精度计数器，详情参见：http：//software. intel. com/sites/products/documentation/hpc/ics/itac/81/ITC _ Reference _ Guide/CPU _ Cycle _ Counter _ （TSC）. htm。——译者注

第 8 章

乘法

8.1 多字乘法

其实完全可以像小学生那样，使用传统的竖式法来计算多字相乘。不过，与其把部分积（partial product）保存在数组里，不如将每次计算好的新行直接放在表示乘积的行中，这样更为高效。

如果被乘数是 m 个字组，而乘数是 n 个字组，那么不论乘法带不带符号，其乘积最多是 $m+n$ 个字组。

在运用“小学竖式法”时，我们把每个 32 位元字组当成一个“数位”（digit）。如果有指令可由两个 32 位整数得出 64 位乘积就好了。不巧的是，即便电脑有这种指令，大多数高级编程语言也无法使用它。实际上，当前很多 RISC 架构的 CPU 都没有这条指令，部分原因是无法从高级编程语言中调用它，从而导致此功能不常用。（还有个原因是，它需要用双寄存器来保存结果，这种指令为数不多。）

图 8.1 列出了算法。它把“半字”视为一个“数位”。参数 w 保存结果，u 和 v 分别是乘数与被乘数。三者都是元素为半字的数组，首个半字（w [0]、u [0]、v [0]）是最低有效数位（least significant digit）。这种数字保存方法是“小端序”的。参数 m 与 n 分别表示 u、v 中的半字个数。

下图也许有助于理解此算法。m 与 n 无关，且不在乎两者谁大谁小。

$$
\begin{array}{r}
u_{m-1}\,u_{m-2}\cdots\cdots\,u_1\;u_0 \\
\times\, v_{n-1}\cdots\,v_1\;v_0 \\
\hline
w_{m+n-1}\;w_{m+n-2}\cdots\cdots\cdots\,w_1\;w_0
\end{array}
$$

计算过程依据［Knu2，4.3.1］所述的“算法 M”（Algorithm M），不过本书使用 C 代码实现，而且将其改为带符号乘法。注意图 8.1 上半部分对 t 的赋值不可能溢出，因为可能赋给 t 的最大值是 $(2^{16}-1)^2+2(2^{16}-1)=2^{32}-1$。

操作数为无符号数的多字乘法最简单了。实际上只要把图 8.1 的“修正”步骤（也就是 3 行注释与返回语句之间的那些代码）省去，就是无符号乘法。将无符号版算法扩展为带符号版，可用如下 3 种方式：

1. 用每个操作数的绝对值做无符号乘法，若两输入值的正负号不同，则给乘积添上负号。

2. 除了高权重的半字外，其余步骤执行无符号数基本乘法，在涉及高权重的半字时，使用“带符号数×无符号数”或“带符号数×带符号数”乘法。

3. 执行无符号乘法并设法修正运算结果。

```
void mulmns(unsigned short w[], unsigned short u[],
   unsigned short v[], int m, int n) {
   unsigned int k, t, b;
   int i, j;

   for (i = 0; i < m; i++)
      w[i] = 0;

   for (j = 0; j < n; j++) {
      k = 0;
      for (i = 0; i < m; i++) {
         t = u[i]*v[j] + w[i + j] + k;
         w[i + j] = t;          // (I.e., t & 0xFFFF).
         k = t >> 16;
      }
      w[j + m] = k;
   }

   // Now w[] has the unsigned product. Correct by
   // subtracting v*2**16m if u < 0, and
   // subtracting u*2**16n if v < 0.

   if ((short)u[m - 1] < 0) {
      b = 0;                    // Initialize borrow.
      for (j = 0; j < n; j++) {
         t = w[j + m] - v[j] - b;
         w[j + m] = t;
         b = t >> 31;
      }
   }
   if ((short)v[n - 1] < 0) {
      b = 0;
      for (i = 0; i < m; i++) {
         t = w[i + n] - u[i] - b;
         w[i + n] = t;
         b = t >> 31;
      }
   }
   return;
}
```

图 8.1 带符号的多字整数乘法

第一种方式需要处理输入的半字，最多要计算 $m+n$ 次绝对值。如果两操作数一正一负，那么，为了处理输入的负值并调整最后的乘积，这种方法最多需要对 $\max(m, n)+m+n$ 个半字执行“求相反数”操作。更严重的地方也许在于，此算法有时必须要先修改其输入值（假设这些值是“按地址传递”的）才能往下计算，而在某些应用场景下，也许不允许这么做。如果不这样做，那么就要为其分配临时空间，或先修改它们，稍后再将其值复原。这几种替代方案都不怎么好。

第二种方式需要3类基本乘法操作（“无符号数×无符号数”、“无符号数×带符号数”、“带符号数×无符号数”），而且需要对部分积执行符号扩展操作，也就是向左填充

“0”或“1”，这样的话，就要花费更长的时间来计算每个部分积，并将其加总到最终乘积。

我们选用第三种方式。为了说明其原理，设 u 与 v 是两个要相乘的带符号整数，分别用 M 与 N 表示其各自位元数。可以看出，图8.1上半部分错把“u”当成了无符号数“$u+2^M u_{M-1}$”，其中 u_{M-1} 指 u 的符号位。也就是说，如果 u 是负数，那么 $u_{M-1}=1$，否则 $u_{M-1}=0$。同理，程序也错把“v”当成了“$v+2^M v_{M-1}$”。

程序计算的是两个无符号数之积，也就是说，它计算的是：

$$(u+2^M u_{M-1})(v+2^N v_{N-1})=uv+2^M u_{M-1}v+2^N v_{N-1}u+2^{M+N}u_{M-1}v_{N-1}$$

为了获得正确结果（也就是 uv），必须从无符号乘积中减去“$2^M u_{M-1}v+2^N v_{N-1}u$”。因为我们知道计算结果可用 $M+N$ 个二进制位来容纳，所以不需要在乘积中考虑其权重比 $M+N-1$ 号位还大的那些位元，于是，也就不用再减掉“$2^{M+N}u_{M-1}v_{N-1}$”这一项了。在图8.1的3行注释之后的代码中，执行了两次减法。它们需要处理最多 $m+n$ 个半字。

大家也许想把全字整数数组（array of fullword integer）传递给图8.1，也就是想跨过两个半字之间的界限，把它们当成一个“全字”来用（lying across the interface）。这种用法在“小端序”计算机中有效，但不适用于“大端序”。如果反向排列数组，也就是把权重最高的半字放在 u[0]（并相应修改程序），那么“跨越式调用法”就能在“大端序”计算机上生效了，不过这回它又不适用于“小端序”了。

8.2 64位积的高权重部分

对于两个32位元整数的64位乘积来说，我们思考如何用程序算出高权重的32位。这也是基本RISC指令 mulhs（multiply high signed，取两个带符号数乘积的高权重部分）与 mulhu（multiply hign unsigned，取两个无符号数乘积的高权重部分）的功能。

如果想获得无符号乘法的高半个乘积，那么完全可以拿图8.1的前半部分来实现算法。将它改写为 $m=n=2$ 时的特例，并把循环展开，把明显无用的操作化简掉，并把参数改为32位无符号整数，这样就好了。

对于带符号乘法来说，并不是非得执行图8.1后半部分的“修正步骤”不可。只要留心中间结果是带符号还是无符号，并适当处理（将变量声明为带符号数，以使右移操作在移位时传播符号位），即可省去修正步骤了。算法代码列在图8.2中。把所有声明为 int 型的数都改成 unsigned 型，即可实现无符号版的算法了。

```
int mulhs(int u, int v) {
   unsigned u0, v0, w0;
   int u1, v1, w1, w2, t;

   u0 = u & 0xFFFF;  u1 = u >> 16;
   v0 = v & 0xFFFF;  v1 = v >> 16;
   w0 = u0*v0;
   t  = u1*v0 + (w0 >> 16);
   w1 = t & 0xFFFF;
   w2 = t >> 16;
   w1 = u0*v1 + w1;
   return u1*v1 + w2 + (w1 >> 16);
}
```

图8.2 求两个带符号数之积的高权重部分

不论是带符号还是无符号，此算法都需16条基本RISC指令，其中4条是乘法。

8.3 无符号与带符号的高权重积互化

假设计算机已经可以根据两个 32 位无符号整数算出其 64 位积的高权重部分，我们想用它来执行带符号数的对应操作。使用图 8.2 可做到这一点，不过那样需要 4 次乘法，[BGN] 给出的方法比它高效很多。

Knuth 的算法 M 可以从无符号版转化为带符号乘法函数（也就是图 8.1），而下面要讲的这个算法，就是上述转化过程的一种特例。设 x 与 y 为两个要相乘的 32 位带符号整数。计算机把 x 视为无符号整数，这样它的值就成了 $x+2^{32}x_{31}$，其中 x_{31} 是 x 的最高有效位（也就是说，若 x 为负，则 x_{31} 是 1，否则是 0）。同理，把 y 看成无符号数后，它就变成了 $y+2^{32}y_{31}$。

虽然我们只想要两数乘积 xy 的高 32 位，不过计算机执行无符号数乘法后算出的值却是：

$$(x+2^{32}x_{31})(y+2^{32}y_{31})=xy+2^{32}(x_{31}y+y_{31}x)+2^{64}x_{31}y_{31}$$

这个值必须减去 $2^{32}(x_{31}y+y_{31}x)+2^{64}x_{31}y_{31}$ 才是想要的答案。由于我们知道结果可以容纳在 64 个位元里，所以把这个结果模 2^{64} 之后，其值不变。这就是说可以直接舍弃最后一项，并按照下列算法计算出带符号数之积的高权重（需要 7 条基本 RISC 指令）。

$$
\begin{array}{ll}
\boldsymbol{p} \leftarrow \mathrm{mulhu}(\boldsymbol{x},\boldsymbol{y}) & // \textit{ multiply high unsigned } \text{instruction} \\
\boldsymbol{t}_1 \leftarrow (\boldsymbol{x} \overset{s}{\gg} 31)\,\&\,\boldsymbol{y} & //\ t_1 = x_{31}y \\
\boldsymbol{t}_2 \leftarrow (\boldsymbol{y} \overset{s}{\gg} 31)\,\&\,\boldsymbol{x} & //\ t_2 = y_{31}x \\
\boldsymbol{p} \leftarrow \boldsymbol{p} - \boldsymbol{t}_1 - \boldsymbol{t}_2 & //\ p = \text{desired result}
\end{array}
\qquad (1)
$$

由带符号的高权重积求无符号的高权重积

反向变换很容易。所用流程与算法 (1) 一样，只是要将第一条指令改为“mulhs”（求带符号数乘积的高权重部分），并把最后一个操作改为 $\boldsymbol{p} \leftarrow \boldsymbol{p}+\boldsymbol{t}_1+\boldsymbol{t}_2$。

8.4 与常数相乘

很容易就能把与常数的乘法操作表示成一系列左移及加法指令。例如，要将 $\boldsymbol{x}$ 乘以 13（二进制 1101），可执行下述操作：

$$
\begin{aligned}
\boldsymbol{t}_1 &\leftarrow \boldsymbol{x} \ll 2 \\
\boldsymbol{t}_2 &\leftarrow \boldsymbol{x} \ll 3 \\
\boldsymbol{r} &\leftarrow \boldsymbol{t}_1 + \boldsymbol{t}_2 + \boldsymbol{x}
\end{aligned}
$$

其中的 $\boldsymbol{r}$ 就是计算结果。

本节中，我们用与 2 的幂相乘来表示左移操作，所以上述算式就写成 $\boldsymbol{r} \leftarrow \boldsymbol{8x} + \boldsymbol{4x} +$

x，这样写可以看出来，在基本RISC及大多数架构的计算机上，此操作需要4条指令。

笔者此处想表达的意思是，除了眼前所见之外，这种操作背后或有玄机。首先，不能单从移位及加法指令的个数上评判某数与常数相乘的算法，还应考虑其他因素。现在演示如下两方案，它们都把一个数乘以45（二进制101101）。

$$
\begin{array}{ll}
t \leftarrow 4x & t_1 \leftarrow 4x \\
r \leftarrow x+t & t_2 \leftarrow 8x \\
t \leftarrow 2t & t_3 \leftarrow 32x \\
r \leftarrow r+t & r \leftarrow t_1+x \\
t \leftarrow 4t & t_3 \leftarrow t_3+t_2 \\
r \leftarrow r+t & r \leftarrow r+t_3
\end{array}
$$

左边的方案用变量 t 保存 x 左移一定长度之后的值，而 x 每次左移的长度则与乘数里面“1”的位置相关。每次移位后的值都是基于上一次移位的结果推算而来。这种方法有三个好处：

- 除了保存输入值用的 x 及保存结果用的 r 外，只需1个寄存器。
- 除开头两步外，只需使用“包含两个地址的指令”（2-address instruction）就够了。
- 移位量相对较小。

对任何乘数来说，左边的方案都具备上述三条属性。

右边的方案用 x 当操作数，把所需执行的全部移位操作先做好。这样的好处是其并发执行力高。只要计算机有足够的指令级并发能力，那么右侧这种方案只需3个周期，而左侧方案在并发能力不受限的电脑中，则需4个周期。

除了上述细节外，要想知道与常数相乘所需的最少操作次数可不那么容易，这里“操作”指的是：由“加法”（add）与“移位”（shift）指令所构成的典型计算机指令集中的指令。在下文中，假定此指令集里有“加法”、“减法”、移位量为常数的“左移”、“取相反数”这4个操作。假定指令格式是“三地址式”（three-address）。将指令集缩减为只能用加法而不能用移位（将某数与其自身相加，然后再把求和结果反复相加，即可实现“左移任意位置”的功能），或将之扩充为可用一条指令实现左移与加法（也就是用一条指令算出 $z \leftarrow x+(y \ll n)$），都不能简化此问题。此外，还需假定我们只需要乘积里面低权重的那32位。

在上述“二进制分解法”（binary decomposition scheme）中，首先可以改进的地方是：当乘数中连续出现至少3个“1”时，可用减法来缩短指令序列。例如，要乘以28（二进制11100），应按 $32x-4x$ 来算（需要3条指令），而不要按 $16x+8x+4x$ 去算（需要5条指令）。在使用“2补码”（two's-complement）的计算机中，即便中间值 $32x$ 溢出，最终计算结果（模 2^{32} 之后）也正确。

用“二进制分解法”与常数 m 相乘（也就是只用移位和加法），需要

$$2\text{pop}(m)-1-\delta$$

条指令，其中若 m 最右侧二进制位是“1”（也就是说 m 为奇数），则 $\delta=1$，否则 $\delta=0$。如果还允许用减法，则需

$$4\mathrm{g}(m)+2\mathrm{s}(m)-1-\delta$$

条指令，其中 $\mathrm{g}(m)$ 是 m 中长度至少为 2 的“全 1 位段”数量，$\mathrm{s}(m)$ 表示 m 中有多少个“孤立的 1”（“singleton” 1-bit），而 δ 含义同上。

如果常量里只包含长度为 2 的“全 1 位段”，则两种方式所需指令数一样。

第二个可改进的地方是，用特殊方式处理由“0”所区隔的位段。例如，考虑 $m=55$（二进制 11 0111）。如果分组计算，那么就是 $(64\boldsymbol{x}-16\boldsymbol{x})+(8\boldsymbol{x}-\boldsymbol{x})$，这需要 6 条指令。而按照 $64\boldsymbol{x}-8\boldsymbol{x}-\boldsymbol{x}$ 来算，则只需 4 条。同理，如果乘以二进制数 1 1011 1011，那么应按照 $512\boldsymbol{x}-64\boldsymbol{x}-4\boldsymbol{x}-\boldsymbol{x}$ 这个公式来算（需要 6 条指令）。

这种公式给出了计算变量 $\boldsymbol{x}$ 乘以任意给定常数 m 所需的最小操作次数。而最大操作次数则基于 m 中的位元个数，也就是 $n=\lfloor \log_2 m \rfloor+1$。

定理 *用“加法”、“减法”、“左移任意个给定位置”这三种操作来计算变量 $\boldsymbol{x}$ 与含有 n 个二进制位的常数 m（$m\geqslant1$）之积，最多需 n 条指令。*

证明（对 n 运用数学归纳法）：与 1 相乘，需要 0 条指令，故当 $n=1$ 时定理成立。在 $n>1$ 时，若 m 以二进制“0”结尾，则与 m 相乘可以化为：先用 $n-1$ 条指令与由 m 左侧 $n-1$ 个二进制位构成的那个数（也就是 $m/2$）相乘，然后再将结果左移 1 位。这共需 n 条指令。

若 m 以二进制“01”结尾，则先用 $n-2$ 条指令把 $\boldsymbol{x}$ 与由 m 左侧 $n-2$ 位所构成的数相乘，然后把结果左移 2 位，再加 $\boldsymbol{x}$。以此计算 $m\boldsymbol{x}$，共需 n 条指令。

若 m 以二进制“11”结尾，则分为“0011”、“0111”、“1011”、“1111”四种情况。设 $\boldsymbol{t}$ 为 $\boldsymbol{x}$ 与由 m 左侧 $n-4$ 位所构成的数之积。若 m 以“0011”结尾，则 $m\boldsymbol{x}=16\boldsymbol{t}+2\boldsymbol{x}+\boldsymbol{x}$，这需要 $(n-4)+4=n$ 条指令。若 m 以“0111”结尾，则 $m\boldsymbol{x}=16\boldsymbol{t}+8\boldsymbol{x}-\boldsymbol{x}$，这需要 n 条指令。若 m 以“1111”结尾，则 $m\boldsymbol{x}=16\boldsymbol{t}+16\boldsymbol{x}-\boldsymbol{x}$，这需要 n 条指令。剩下一种情况就是 m 以“1011”结尾。

而在这种情况下，若 m 以“00 1011”、“01 1011”、“11 1011”结尾，则不难证明 $m\boldsymbol{x}$ 可用 n 条指令算出。剩下一种小情况，就是 m 以“10 1011”结尾。

这种规律会依次延续下去，每次“剩下的那种情况”，都是以“10 1010...1010 1011”的形式结尾。最后，当把 m 的全部二进制位都包括在内，剩下的唯一一种情况就是“10 1010...1010 1011”。在这 n 个二进制位中，值为“1”的位有 $n/2+1$ 个。根据前述“$2\mathrm{pop}(m)-1-\delta$”这个公式，$\boldsymbol{x}$ 与此数相乘，可用 $2(n/2+1)-2=n$ 条指令算出。■

通过上述证明尤其可见：用此方法与 32 位常数相乘，最多需 32 条指令。逐个检验可发现，当 n 为偶数时，与 n 位二进制数 10 1010...10 1011 相乘，需要 n 条指令，而当 n 为奇数时，与 n 位二进制数 10 1010...01 0110 相乘，也需 n 条指令，所以这个边界是“紧界”，n 的值不能再缩小了。

上面描述的整套计算方式不难用手算实现，而且，想把它整理为算法并集成到编译

器中也较为容易；不过，这种算法未必总能产生最优代码，因为有时还能继续改进。在计算 mx 的过程中，还可以对乘数 m 或某个中间量进行因数分解。仍以 $m=45$（二进制数 10 1101）为例。按上述方案需要 6 条指令。如果将 45 分解为 $5 \cdot 9$，那么只用 4 条就够了：

$$
\begin{aligned}
t &\leftarrow 4x+x \\
r &\leftarrow 8t+t
\end{aligned}
$$

因数分解也可以和二进制分解法合起来用。例如，按二进制分解法，乘以 106（二进制 110 1010）需要 7 条指令，但将其写为 $7 \cdot 15+1$，则只需 5 条指令。对于大常数来说，乘法所需最少指令数也许远比单用上述二进制分解法要小。例如，按二进制分解法，$m=$ 0xAAAA AAAB 需要 32 条指令，但若将其写为 $2 \cdot 5 \cdot 17 \cdot 257 \cdot 65\,537+1$ 则只需 10 条。（对于大常数来说，10 条指令也许不是常见情况。这种因数分解表明，此数有较为简单的“位模式”（bit pattern），其中“1”和“0”交替出现。）

似乎并没有简单的公式或流程能判定与常数 m 相乘最少需要几条移位及加法指令。[Bern] 给出了较为实用的搜索流程，不过它并非总能找到最小值。可以设计出穷举法来找寻最小值，不过其空间与时间开销都很高。（例如，可参考 [Knu2, 4.6.3] 中图 15 所列的树状结构。）

这个看似简单的问题似乎涉及组合学。高德纳教授在 [Knu2, 4.6.3] 中讨论了一个与之密切相关的问题：计算 a^m 最少要用几次乘法。它与“只用加法指令实现与常数 m 相乘”这一问题类似。

8.5 习题

1. 证明：在 $32 \times 32 \Rightarrow 64$ 位乘法中，无论将操作数视为带符号整数或无符号整数，其乘积的低权重 32 位都一样。
2. 修改图 8.2 中的 mulhs 函数，令其在计算 64 位积的高权重部分时，也能计算出其低权重部分。（只修改计算过程，不涉及参数传递问题。）
3. 复数乘法可定义如下：

$$(a+b\mathrm{i})(c+d\mathrm{i})=ac-bd+(ad+bc)\mathrm{i}$$

只用 3 次乘法即能实现此运算[⊖]。设

$$
\begin{aligned}
p&=ac \\
q&=bd \\
r&=(a+b)(c+d)
\end{aligned}
$$

则两复数之积就是：

⊖ 据传此方法源自高斯（Gauss）。

$$p-q+(r-p-q)\mathrm{i}$$

读者很容易验证此算法。

编写类似算法，只用 3 次乘法指令求出两个 32 位无符号整数的 64 位乘积。假定计算机的乘法指令可算出两个 32 位整数之积的低权重 32 位（不论是带符号还是无符号数，这样乘出来的结果相同）。

第 9 章

整数除法

9.1 预备知识

本章及下一章要讲一些涉及“计算机整数除法”（“computer division” of integers）的算法。在数学公式里，我们用 x/y 表示普通的有理数除法，用 $x \div y$ 表示带符号的计算机整数除法（向 0 取整，trucating toward 0），用 $x \overset{u}{\div} y$ 表示无符号的计算机整数除法。在 C 语言代码中，x/y 的意思当然是执行计算机除法了，只要其中一个操作数是无符号数，就执行无符号除法，两个操作数皆为带符号数时，执行带符号除法。

除法是个复杂过程，涉及它的算法通常都不太优雅，甚至对带符号整数除法的定义都有不同解读。大多数高级语言以及大部分计算机指令集，都把有理数除法的结果向 0 取整后的值，定义为带符号整数除法之商。下面演示了这种取整方式及另外两种取整方式。

		向 0 取整式除法（截取式除法）	余数非负式除法（“模”除法）	向下取整式除法（地板式除法）
$7 \div 3$	=	商 2 余 1	商 2 余 1	商 2 余 1
$(-7) \div 3$	=	商 −2 余 −1	商 −3 余 2	商 −3 余 2
$7 \div (-3)$	=	商 −2 余 1	商 −2 余 1	商 −3 余 −2
$(-7) \div (-3)$	=	商 2 余 −1	商 3 余 2	商 2 余 −1

这三种取整方式都遵从“被除数 = 商 * 除数 + 余数”这一关系式。“模”除法（“modulus” division）的意思是保持余数非负[⊖]。“地板式”除法（“floor” division）的意思是要对有理数除法的商向下取整。在除数为正的情况下，“模”除法与“地板式”除法等效。还有一种很少用到的除法方式，那就是将商向最近的整数取整。

“余数非负式除法”与“向下取整式除法”有个好处：大多数技巧若用它们来描述，

⊖ 笔者所用的命名法也许会受人指摘，因为大家并未普遍认同“模”（modulus）一词隐含了“非负”（nonnegative）的意思。高德纳教授把“模”（mod）操作符［Knu1］定义为求“地板式除法”的余数，若除数是负数，则余数是负数或 0。许多计算机程序语言用“mod”表示截取式除法的余数。然而，在数学里，“模”这个词有时表示复数的“幅值”（the magnitude of a complex，即实部与虚部各自平方之和的开方），这个值非负，而在“同余理论”（congruence theroy）中，则通常假定此值为正。

则会变得简单一些。例如，除以 2^n 可以改用带符号右移 n 位来实现，除以 2^n 的余数可以通过对 x 和 2^n-1 求“逻辑与”来计算。笔者估计，采用“余数非负式除法”与“向下取整式除法”，恐怕更容易得出想要的结果。比方说，要编写程序来绘制整数值函数（integer-valued function）曲线图，函数值的范围在 *imin* 至 *imax* 之间。现在要找出间距最小的两数作为图中纵坐标的两个极值，它们必须是 10 的倍数，且能将 *imin* 与 *imax* 纳于其间。如果用“余数非负式除法”或“向下取整式除法”，那么这两个极值很好计算，只需分别用 $(imin \div 10) * 10$ 及 $((imax+9) \div 10) * 10$ 即可求出。如果改为惯用的除法，那么必须依照类似下列方式来求值：

```
if (imin >= 0) gmin = (imin/10)*10;
else           gmin = ((imin - 9)/10)*10;
if (imax >= 0) gmax = ((imax + 9)/10)*10;
else           gmax = (imax/10)*10;
```

“余数非负式除法”与“向下取整式除法”求出的商，要比“向 0 取整式除法”的商用处更多，此外，笔者推测，与可能为负的余数相比，我们可能更需要非负的余数。

很难在“余数非负式除法”与“向下取整式除法”之间取舍，因为二者仅在除数为负时才有差别，而负数作除数的情况并不常见。此问题也无法参考现有高级编程语言所用的方法，因为它们通常使用“向 0 取整式除法”来计算操作数均为带符号整数的算式 x/y。还有些语言以浮点数或有理数来表示除法结果。如果从余数角度看，各个语言之间的做法比较混乱。Fortran 90 里的 MOD 函数给出“向 0 取整式除法”的余数，而 MODULO 给出“向下取整式除法”的余数（此数可能是负数）。与之类似，Common Lisp 与 ADA 语言也有两种模除：REM 给出“向 0 取整式除法”的余数，而 MOD 给出“向下取整式除法”的余数。在 PL/I 中，MOD 总是非负（也就是说，它给出“余数非负式除法”的余数）。在 Pascal 中，A mod B 只有在 B>0 时才有定义，而且如果有定义，必然给出非负值（此值可理解为“余数非负式除法”或“向下取整式除法”的余数）。

不管怎么说，即便知道如何修改除法的定义，我们也无力改变它[⊖]，所以，在后续文字中，还是把 $x \div y$ 定义成常见的除法操作（也就是“向 0 取整式除法”）吧！

“向 0 取整式除法”有个属性很经典，那就是它满足下列等式：

$$(-n) \div d = n \div (-d) = -(n \div d), \text{for } d \neq 0$$

如果要在程序码中利用该等式来变换除法操作，那么要小心：若 n 或 d 是绝对值最大的负数，则 $-n$ 或 $-d$ 无法用 32 个二进制位来表示。$(-2^{31}) \div (-1)$ 会溢出（其结果无法以“2 补码”形式保存到 32 位带符号整数中），大多数计算机将该操作的结果视为未定义，或禁用此操作。

操作数为带符号整数的“向 0 取整式除法”，与普通的有理数乘法之间，有下述关系：

⊖ 也有人试着改变。例如 IBM 的 PL.8 语言就把 $x \div y$ 定义成“余数非负式除法”，而高德纳的 MMIX 计算机把除法指令定义成“向下取整式除法”[Knu7]。

$$n \div d = \begin{cases} \lfloor n/d \rfloor, \text{若 } d \neq 0, nd \geqslant 0, \\ \lfloor n/d \rfloor, \text{若 } d \neq 0, nd < 0. \end{cases} \tag{1}$$

无符号整数除法，也就将是 n 与 d 皆视为无符号整数的除法，满足（1）中上面那部分。

在接下来的讨论中，将使用下列各式所描述的算术基本性质，此处不加证明。可参看［Knu1］与［GKP］两本书，它们有趣地讲解了向下取整函数（floor function）与向上取整函数（ceiling function）。

定理 D1 对于实数 x 与整数 k，有：

$$\begin{array}{ll}
\lfloor x \rfloor = -\lceil -x \rceil & \lceil x \rceil = -\lfloor -x \rfloor \\
x-1 < \lfloor x \rfloor \leqslant x & x \leqslant \lceil x \rceil < x+1 \\
\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1 & \lceil x \rceil - 1 < x \leqslant \lceil x \rceil \\
x \geqslant k \Leftrightarrow \lfloor x \rfloor \geqslant k & x \leqslant k \Leftrightarrow \lceil x \rceil \leqslant k \\
x > k \Rightarrow \lfloor x \rfloor \geqslant k & x < k \Rightarrow \lceil x \rceil \leqslant k \\
x \leqslant k \Rightarrow \lfloor x \rfloor \leqslant k \Rightarrow x < k+1 & x \geqslant k \Rightarrow \lceil x \rceil \geqslant k \Rightarrow x > k-1 \\
x < k \Leftrightarrow \lfloor x \rfloor < k & x > k \Leftrightarrow \lceil x \rceil > k
\end{array}$$

定理 D2 对于整数 n，d，若 $d>0$，则有：

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d+1}{d} \right\rceil, \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d-1}{d} \right\rfloor$$

若 $d<0$，则有：

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d-1}{d} \right\rceil, \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d+1}{d} \right\rfloor$$

定理 D3 对于实数 x 和整数 d（$d>0$），有：

$$\lfloor \lfloor x \rfloor / d \rfloor = \lfloor x/d \rfloor, \lceil \lceil x \rceil / d \rceil = \lceil x/d \rceil$$

推论 对于实数 a，b（$b \neq 0$），d（$d>0$），有：

$$\left\lfloor \left\lfloor \frac{a}{b} \right\rfloor / d \right\rfloor = \left\lfloor \frac{a}{bd} \right\rfloor, \left\lceil \left\lceil \frac{a}{b} \right\rceil / d \right\rceil = \left\lceil \frac{a}{bd} \right\rceil$$

定理 D4 对于整数 n，d（$d \neq 0$）及实数 x，有：

$$\text{若 } 0 \leqslant x < \left| \frac{1}{d} \right| \text{ 则} \left\lfloor \frac{n}{d} + x \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor, \text{若} - \left| \frac{1}{d} \right| < x \leqslant 0 \text{ 则} \left\lceil \frac{n}{d} + x \right\rceil = \left\lceil \frac{n}{d} \right\rceil$$

在下面列出的定理中，$\mathrm{rem}(n, d)$ 表示 n 除以 d 的余数。若 d 为负数，则 $\mathrm{rem}(n, -d) = \mathrm{rem}(n, d)$，这与"向 0 取整式除法"和"余数非负式除法"的规则相同。我们不在 $n<0$ 时使用 $\mathrm{rem}(n, d)$。因此，在这种用法下，余数总是非负数。

定理 D5 对 $n \geqslant 0$，$d \neq 0$，有：

$$\mathrm{rem}(2n, d) = \begin{cases} 2\mathrm{rem}(n, d) \\ 2\mathrm{rem}(n, d) - |d| \end{cases} \mathrm{rem}(2n+1, d) = \begin{cases} 2\mathrm{rem}(n, d) + 1 \\ 2\mathrm{rem}(n, d) - |d| + 1 \end{cases}$$

（这两个式子求出的值都大于等于 0，且小于 $|d|$）。

定理 D6 对于 $n \geqslant 0$，$d \neq 0$，有：

$$\text{rem}(2n,\ 2d) = 2\text{rem}(n,\ d)$$

定理 D5 与 D6 很容易通过余数的定义来证明，也就是说，对于 $n \geqslant 0$，$d \neq 0$（n 与 d 可以不是整数，不过我们只在整数情况下使用这两个定理），可找到整数 q，使下式成立：

$$n = qd + \text{rem}(n,\ d)\text{，其中 } 0 \leqslant \text{rem}(n,\ d) < |d|$$

9.2 多字除法

与多字乘法相似，也可以用传统的竖式法（grade school method）来计算多字除法。然而，细节却出乎意料地复杂。图 9.1 用 C 代码写出了高德纳（Knuth）教授的"算法 D"（Algorithm D）。其中所用除法的底层形式为 $32 \overset{u}{\div} 16 \Rightarrow 32$。（实际上，这些底层除法操作的商最多占 17 个二进制位。）

```
int divmnu(unsigned short q[], unsigned short r[],
     const unsigned short u[], const unsigned short v[],
     int m, int n) {

   const unsigned b = 65536; // Number base (16 bits).
   unsigned short *un, *vn;  // Normalized form of u, v.
   unsigned qhat;            // Estimated quotient digit.
   unsigned rhat;            // A remainder.
   unsigned p;               // Product of two digits.
   int s, i, j, t, k;

   if (m < n || n <= 0 || v[n-1] == 0)
      return 1;              // Return if invalid param.

   if (n == 1) {                        // Take care of
      k = 0;                            // the case of a
      for (j = m - 1; j >= 0; j--) {    // single-digit
         q[j] = (k*b + u[j])/v[0];      // divisor here.
         k = (k*b + u[j]) - q[j]*v[0];
      }
      if (r != NULL) r[0] = k;
      return 0;
   }

   // Normalize by shifting v left just enough so that
   // its high-order bit is on, and shift u left the
   // same amount. We may have to append a high-order
   // digit on the dividend; we do that unconditionally.

   s = nlz(v[n-1]) - 16;        // 0 <= s <= 16.
   vn = (unsigned short *)alloca(2*n);
   for (i = n - 1; i > 0; i--)
      vn[i] = (v[i] << s) | (v[i-1] >> 16-s);
   vn[0] = v[0] << s;

   un = (unsigned short *)alloca(2*(m + 1));
   un[m] = u[m-1] >> 16-s;
   for (i = m - 1; i > 0; i--)
      un[i] = (u[i] << s) | (u[i-1] >> 16-s);
   un[0] = u[0] << s;
   for (j = m - n; j >= 0; j--) {       // Main loop.
      // Compute estimate qhat of q[j].
      qhat = (un[j+n]*b + un[j+n-1])/vn[n-1];
      rhat = (un[j+n]*b + un[j+n-1]) - qhat*vn[n-1];
```

图 9.1 多字无符号整数除法

```
again:
      if (qhat >= b || qhat*vn[n-2] > b*rhat + un[j+n-2])
      { qhat = qhat - 1;
        rhat = rhat + vn[n-1];
        if (rhat < b) goto again;
      }

      // Multiply and subtract.
      k = 0;
      for (i = 0; i < n; i++) {
         p = qhat*vn[i];
         t = un[i+j] - k - (p & 0xFFFF);
         un[i+j] = t;
         k = (p >> 16) - (t >> 16);
      }
      t = un[j+n] - k;
      un[j+n] = t;

      q[j] = qhat;              // Store quotient digit.
      if (t < 0) {              // If we subtracted too
         q[j] = q[j] - 1;       // much, add back.
         k = 0;
         for (i = 0; i < n; i++) {
            t = un[i+j] + vn[i] + k;
            un[i+j] = t;
            k = t >> 16;
         }
         un[j+n] = un[j+n] + k;
      }
   } // End j.
   // If the caller wants the remainder, unnormalize
   // it and pass it back.
   if (r != NULL) {
      for (i = 0; i < n; i++)
         r[i] = (un[i] >> s) | (un[i+1] << 16-s);
   }
   return 0;
}
```

图 9.1 （续）

算法每次处理半字长的输入，并输出半个字组。当然也可以每次处理一个完整的字组，不过那种算法需要做 $64 \overset{u}{\div} 32 \Rightarrow 32$ 的除法。此处假定要么计算机没有此指令，要么很难在高级语言中调用它。尽管我们通常假定计算机执行 $32 \overset{u}{\div} 32 \Rightarrow 32$ 除法，不过对于多字除法来说，$32 \overset{u}{\div} 16 \Rightarrow 16$ 足够了。

因此，对于 Knuth 算法的此种实现代码来说，基数（base）b 是 65 536。此算法的大部分解释都在［Knu2］一书中。

被除数 u 与除数 v 都以“小端序”存储，也就是说，u[0] 与 v[0] 分别存放两数的低权重数位。（此代码在“大端序”与“小端序”的计算机上均适用。）参数 m 与 n 分别表示 u 与 v 中的半字个数（高德纳把 m 定义为商的长度）。算法的调用者负责为商 q 准备存储空间，此外也可以给余数 r 分配空间。至少需要 m - n + 1 个半字来存放商，而余数 r 则需要 n 个半字。如果不关心余数，那么可以把接收余数所用的地址填成 NULL。

算法要求除数的最高有效数位（也就是 v[n - 1]）不得为 0。这简化了规范化（normalization）的步骤，而且有助于开发者确认已为商分配了足够的空间。算法代码要

判断 v[n-1] 是否为非 0 值，同时还要确保 n≥1 及 m≥n。若任何一项条件不满足，则返回错误代码（返回的错误代码为 1）。

执行完这些检测后，代码首先判断除数长度是否为 1，如果是，就执行这种最简单的除法。将此情况单列出来，不是为了提高算法速度，而是因为算法其余部分要求除数长度至少是 2。

如果除数长度大于等于 2，那么算法会将其规范化，也就是把它左移足够长度，使得最高权重的位元变成 1。而被除数也要移动同样长度，这样商就不受移位操作影响了。高德纳解释说：这些步骤很值得，因为执行之后，我们可以把余数的每个数位猜得更加准确。前导 0 计数函数（也就是 nlz($\boldsymbol{x}$)）用于决定移位量。

在规范化步骤中，算法给规范后的被除数与除数分配了存储空间。这么做一来是因为算法调用者通常不希望改变输入的参数值，二来是因为有些输入值可能是存放于只读内存（read-only memory）中的常数，无法改变。此外，可能还需要给被除数添加一个高权重数字。C 语言的“alloca”函数很适合分配这种临时空间。此函数通常实现得很高效，只需 2 至 3 条内联指令（in-line instruction）即可分配空间，而释放空间则无需任何指令。空间分配在程序的栈上，这样的话，当子程序返回时，即可自动释放空间。

在主循环里，每一轮都算出商里的一个数字，而被除数则不断缩减，直至变为余数。求商过程中，每个数位上估算出来的数字都放在 qhat 中，由 again 标签所引领的那段程序可将其完善，这样的话，估算值要么刚好准确，要么比正确值大 1。

下面几个步骤把 qhat 乘以除数，然后将乘积从当前的余数中减去，这一点与竖式除法相同。若余数为负，则必须把商里的当前数位减 1，再重新计算乘法与减法，更简单的办法是，直接把除数加到当前的余数上，这样就把余数重新调整好了。这个调整过程最多只需 1 次，因为商里的当前数字要么刚好准确，要么比正确值大 1。

最后，如果用来存放余数的地址不是 null，那么就把余数经此传给调用者。余数必须右移 s 位，这个 s 是被除数与除数在规范化步骤中的移位量。

“加回”（add back）这一步很少会执行，因为我们可以看到，算法初次计算商里每个位置上的数字 qhat 时，使用当前余数里权重最高的两个数字除以除数里权重最高的一个数字。“again”循环中的步骤完善了 qhat 的值，把它设为当前余数中权重最高的三位数字与除数中权重最高的两位数字之商（证明过程略去。读者可以假设 b=10，也就是用十进制数验证一下这一方法）。请注意，由于经过规范化处理，所以每轮的除数必然都大于等于 b/2，且每轮的被除数都小于等于 b 和除数之积（因为余数小于除数）。

如果估算商时只用被除数前三个数字除以除数前两个数字，那么效果如何呢？由于执行了规范化，因此可以证实，这种估算方式相当精确。为了较为直观地说明（不是严格证明）其道理，我们考虑以此方式执行十进制数除法 u/v。可以看出，估算值总是高于（或恰好等于）正确值。因此，最坏情况就是：代表除数的那两位数字最大限度地缩减了

除数本来的值，而代表被除数的那三位数字却几乎没有缩减其本来的值（也就是说这三位数字就能代表被除数本身），同时，被除数要尽可能大，这三条合起来，可以最大限度地放大误差。在 49900...0/5099...9 时，就会出现这种情况，此时估算出来的商是 499/50＝9.98，正确结果约为 499/51≈9.7843。由于两者之差是 0.1957，而估算的商数字与正确的商数字分别要对前两个比值向下取整，所以可见，这两个数值之间最多差 1，而这种情况大约有 20％的几率发生（假设商里各个位置上的数字是均匀分布的）。这反过来意味着“加回”步骤大约有 20％的几率要执行。

将这个（不严格的）分析运用于以 b 为基的情况，可以得出，估算的商与真实的商之间最多差 $2/b$。对于 $b=65\,536$ 的情况来说，仍然可以得出估算的商数字与真实的商数字之间最多差 1，而这种情况发生的概率是 2/65 536≈0.000 03。因此，用“加回”步骤调整商数字的概率只有大约 0.003％。

在十进制数除法中，需要执行“加回”步骤的例子是 4500/501。而在 65 536 进制下，类似的例子是 0x7FFF 8000 0000 0000/0x8000 0000 0001。

本书不打算估计整个程序的执行时间，不过显然可以看出，在 m 与 n 值较大时，执行时间由“乘法/减法循环”（multiply/subtract loop）控制。优秀的编辑器可以把此循环编译成大约 16 条基本 RISC 指令，其中 1 条是乘法。循环控制器为 j 的 for 循环要执行 $m-n+1$ 次，而“乘法/减法循环”要执行 n 次，那么这部分程序的执行时间就是（$15+mul$）n（$m-n+1$）个周期，其中 mul 是将两个 16 位元的变量相乘所需的时间。程序还需执行（$m-n+1$）次除法指令以及 1 次前导 0 计数指令。

带符号多字除法

本书不会专门给出带符号版的多字除法算法，只是告诉大家，根据以下步骤即可将无符号版算法改编为带符号版：

1. 若被除数为负，则将之设为其相反数，除数也依此处理。
2. 将被除数与除数改用无符号数表示。
3. 使用无符号多字除法算法。
4. 将商和余数改用带符号数表示。
5. 若被除数与除数正负号不同，则将商设为其相反数。
6. 若被除数为负，则对余数取相反数。

这些步骤有时需要增加或删除一个最高有效数位。为了简单一些，我们以 256 进制数来举例（也就是每个字节表示一个数字），而且假设数字序列里权重最高的位元表示符号位。这很像常见的“2 补码”表示法。考虑 255 为除数的情况，它用带符号数写出来是 0x00FF。在第 2 步中，它必须缩减成 0xFF。与之类似，如果从第 3 步中得出的商以值为“1”的位元开头，那么为了正确将其改用带符号数表示出来，就必须在它前面加上一个 0 值字节。

9.3 用带符号除法计算无符号短除法

"短除法"（short division）的意思是，用一个单字除以另一个单字（例如：32÷32⇒32）。在 C 语言及其他很多高级语言里，使用"/"操作符，并以整数为操作数，即可实现此种形式的除法。C 语言支持带符号短除法与无符号短除法，但是有些计算机的指令集里只有带符号的除法指令。那么，如何在此类计算机上实现无符号除法呢？似乎没有巧妙的办法，不过本书提供几个可选方案。

9.3.1 用带符号长除法计算无符号短除法

即便计算机有带符号长除法（64÷32⇒32），以此实现无符号短除法也没有想象的那么容易。IBM RS/6000 的 XLC 编译器用下列步骤实现 $\boldsymbol{q} \leftarrow (\boldsymbol{n} \overset{u}{\div} \boldsymbol{d})$：

$$\begin{aligned}
&\text{if } \boldsymbol{n} \overset{u}{<} \boldsymbol{d} \text{ then } \boldsymbol{q} \leftarrow \mathbf{0} \\
&\text{else if } \boldsymbol{d} = \mathbf{1} \text{ the } \boldsymbol{q} \leftarrow \boldsymbol{n} \\
&\text{else if } \boldsymbol{d} \leqslant \mathbf{1} \text{ the } \boldsymbol{q} \leftarrow \mathbf{1} \\
&\text{else } \boldsymbol{q} \leftarrow (\mathbf{0} \parallel \boldsymbol{n}) \div \boldsymbol{d}
\end{aligned}$$

其中第 3 行实际上是想判断 $\boldsymbol{d} \overset{u}{\geqslant} \mathbf{2}^{31}$。如果此处 $\boldsymbol{d}$ 作为带符号数，其字面值小于等于 1，那么，由于它不可能为 1（由第二行已经排除），所以其字面值必然小于等于 0。不考虑 $\boldsymbol{d}=\mathbf{0}$ 的情况，对于有意义的情况来说，如果第三行这条测试语句是 ***true***，那么 $\boldsymbol{d}$ 的符号位就是"1"，因此，$\boldsymbol{d} \overset{u}{\geqslant} \mathbf{2}^{31}$。由于从第一行可知 $\boldsymbol{n} \overset{u}{\geqslant} \boldsymbol{d}$，而且 $\boldsymbol{n}$ 不可能超过 $\mathbf{2}^{32}-\mathbf{1}$，所以 $\boldsymbol{n} \overset{u}{\div} \boldsymbol{d} = \mathbf{1}$。

第 4 行这种写法的意思是用 32 个"0"后面跟着 32 位二进制数 $\boldsymbol{n}$ 构成一个双字长的整数，并将它除以 $\boldsymbol{d}$。必须在第 2 行测试 $\boldsymbol{d}=\mathbf{1}$，以确保此除法不会溢出（如果 $\boldsymbol{n} \overset{u}{\geqslant} \mathbf{2}^{31}$，而又不在第 2 行测试，那么除法就会溢出，这时商未定义）。

合并第 2 行及第 3 行的比较操作[㊀]，实现上述算法需要 11 条指令，其中 3 条是分支指令。如果需要在 $\boldsymbol{d}=\mathbf{0}$ 时执行除法操作，并产生溢出中断，那么可把第 3 行改为"else if $\boldsymbol{d}<\mathbf{0}$ then $\boldsymbol{q} \leftarrow \mathbf{1}$"，这样的话，在 RS/6000 计算机上需要 12 条指令。

如果不想对或许比较常见的情况（$\mathbf{2} \overset{u}{\leqslant} \boldsymbol{d} \overset{u}{\leqslant} \mathbf{2}^{31}$）执行太多测试，那么要修改上述代码也很容易（把 $\boldsymbol{d} \leqslant \mathbf{1}$ 放在开头测试就好），只是代码量稍微增加了一点。

9.3.2 用带符号短除法计算无符号短除法

本节为 32 位计算机而写，不过，只要把其中的"31"全部改成"63"，就可适用于

㊀ RS/6000 计算机只需执行一次比较指令即可设置多个状态位，以表示小于、大于、等于关系。

64 位机（也就是说，使用同样形式的带符号除法，亦可算出无符号 $64 \div 64 \Rightarrow 64$ 除法）。此算法可用在 Java 这种缺乏无符号整数的编程语言中。

如果带符号长除法不可用，而带符号短除法可用，那么就可以设法把问题化简为 n，$d<2^{31}$ 的情况，然后使用计算机上的除法指令来实现 $\boldsymbol{n} \overset{u}{\div} \boldsymbol{d}$。若 $\boldsymbol{d} \overset{u}{\geqslant} \mathbf{2}^{\mathbf{31}}$，则 $\boldsymbol{n} \overset{u}{\div} \boldsymbol{d}$ 不是 **0** 就是 **1**，于是可以先抛开这种简单的情况。用 $(\boldsymbol{n} \overset{u}{\div} 2) \div \boldsymbol{d}) \times 2$ 可以估算 $\boldsymbol{n} \overset{u}{\div} \boldsymbol{d}$，结果即便不对，离正确值也只差 1，据此，我们可缩减被除数。此算法可以归结如下：

1. if $\boldsymbol{d}<\mathbf{0}$ then if $\boldsymbol{n} \overset{u}{<} \boldsymbol{d}$ then $\boldsymbol{q} \leftarrow \mathbf{0}$
2.   else $\boldsymbol{q} \leftarrow \mathbf{1}$
3. else do
4. $\boldsymbol{q} \leftarrow ((\boldsymbol{n} \overset{u}{\div} 2) \div \boldsymbol{d}) \times 2$
5. $\boldsymbol{r} \leftarrow \boldsymbol{n}-\boldsymbol{q d}$
6.   if $\boldsymbol{r} \overset{u}{\geqslant} \boldsymbol{d}$ then $\boldsymbol{q} \leftarrow \boldsymbol{q}+\mathbf{1}$
7. end

第 1 行判断 $\boldsymbol{d}<\mathbf{0}$ 实际是为了测试 $\boldsymbol{d} \overset{u}{\geqslant} \mathbf{2}^{\mathbf{31}}$。如果 $\boldsymbol{d} \overset{u}{\geqslant} \mathbf{2}^{\mathbf{31}}$，那么商的最大值就是 $(2^{32}-1) \div 2^{31}=1$，于是前两行算法可以正确计算出此情况下的商。

第 4 行所表示的代码会将被除数右移一位，除以除数，然后再左移一位。此时显然有 $\boldsymbol{n} \overset{u}{\div} 2 \overset{u}{<} 2^{31}$，而 $\boldsymbol{d} \overset{u}{<} \mathbf{2}^{\mathbf{31}}$ 也成立，所以这些数都可以当成计算机带符号除法指令的操作数。（若 $\boldsymbol{d}=\mathbf{0}$，则产生溢出信号。）

第 4 行估算出来的值是：

$$q=\lfloor\lfloor n/2 \rfloor/d \rfloor \cdot 2=\lfloor n/(2d)\rfloor \cdot 2=\frac{n-\operatorname{ren}(n,2d)}{d}$$

其中用到了定理 D3 的推论。第 5 行根据估算出来的商计算余数。其值为：

$$r=n-\frac{n-\operatorname{rem}(n,2d)}{d} d=\operatorname{rem}(n,2d)$$

因此，$0 \leqslant r<2d$。若 $r<d$，则 q 就是正确的商。若 $r \geqslant d$，则 q 加 1 就是正确的商（此处程序必须执行无符号比较，因为 r 有可能大于等于 2^{31}）。

如果在比较 $\boldsymbol{n} \overset{u}{<} \boldsymbol{d}$ 之前，先使用常数加载指令把 **0** 装入 $\boldsymbol{q}$ 中，并将第 2 行 $\boldsymbol{q} \leftarrow \mathbf{1}$ 的情况并入第 6 行 $\boldsymbol{q} \leftarrow \boldsymbol{q}+\mathbf{1}$ 这一赋值语句中，那么在大部分计算机中此算法需要 14 条指令，其中 4 条是分支指令。很容易就能扩充代码功能，使之同时算出余数：在第 1 行后追加 $\boldsymbol{r} \leftarrow \boldsymbol{n}$，在第 2 行后添上 $\boldsymbol{r} \leftarrow \boldsymbol{n}-\boldsymbol{d}$，并且在第 6 行的“then”子句后面补上 $\boldsymbol{r} \leftarrow \boldsymbol{r}-\boldsymbol{d}$。（也可以不采用上述改法，而是在整个算法序列最后加上 $\boldsymbol{r} \leftarrow \boldsymbol{n}-\boldsymbol{q d}$，这样要计算一次乘法。）

第 1、2 行还可写成：

if $\boldsymbol{n} \overset{u}{<} \boldsymbol{d}$ then $\boldsymbol{q} \leftarrow \mathbf{0}$
else if $\boldsymbol{d}<\mathbf{0}$ then $\boldsymbol{q} \leftarrow \mathbf{1}$

这样代码能稍微精简一点，共需 13 条指令，其中 3 条是分支指令。不过它在通常情况下（n，d 都较小且 $n>d$）会执行更多条指令。

若采用谓词表达式，则程序可写为：

$$
\begin{array}{ll}
1. & \text{if } \boldsymbol{d}<\boldsymbol{0} \text{ then } \boldsymbol{q}\leftarrow(\boldsymbol{n}\overset{u}{\geqslant}\boldsymbol{d})\\
2. & \text{else do}\\
3. & \quad \boldsymbol{q}\leftarrow((\boldsymbol{n}\overset{u}{\div}2)\div\boldsymbol{d})\times 2\\
4. & \quad \boldsymbol{r}\leftarrow\boldsymbol{n}-\boldsymbol{qd}\\
5. & \quad \boldsymbol{q}\leftarrow\boldsymbol{q}+(\boldsymbol{r}\overset{u}{\geqslant}\boldsymbol{d})\\
6. & \text{end}
\end{array}
$$

如果不用分支就能算出这些谓词，那么上述方式能省两条分支指令。在基本 RISC 架构计算机中，用一条指令（CMPGEU）就能求出谓词的值；在 MIPS 架构上，用两条（SLTU，XORI）。如果在计算上述程序第 1 行中的谓词时根据 $\boldsymbol{d}_{31}=1$ 化简，在计算第 5 行的谓词时根据 $\boldsymbol{d}_{31}=0$ 化简，那么，大多数计算机采用 2.12 节“比较谓词”中计算 $\boldsymbol{x}\overset{u}{\leqslant}\boldsymbol{y}$ 的方法求每个谓词，需要 4 条指令（如果包含全套逻辑指令，那就是 3 条）。这两个谓词表达式可化简为：

$$\text{第 1 行：} \boldsymbol{n}\overset{u}{\geqslant}\boldsymbol{d}=(\boldsymbol{n}\ \&\neg(\boldsymbol{n}-\boldsymbol{d}))\overset{u}{\gg}31$$

$$\text{第 5 行：} \boldsymbol{r}\overset{u}{\geqslant}\boldsymbol{d}=(\boldsymbol{r}\ |\neg(\boldsymbol{r}-\boldsymbol{d}))\overset{u}{\gg}31$$

如果在 $\boldsymbol{d}\overset{u}{\geqslant}\boldsymbol{2^{31}}$ 时把被除数强制设为 $\boldsymbol{0}$，那么就可写出无分支的代码。这样一来，除数就可以放在计算机的带符号除法指令中了，因为把它错误地视为负数后，商为 0，这与正确结果之间的误差不会超过 1。在被除数很大的情况下，仍然要将其右移一位之后再除以除数，除法求出的商要左移一位才行。这样写出来的程序是（10 条基本 RISC 指令）：

$$
\begin{array}{l}
\boldsymbol{t}\leftarrow\boldsymbol{d}\overset{s}{\gg}31\\
\boldsymbol{n}'\leftarrow\boldsymbol{n}\ \&\neg\boldsymbol{t}\\
\boldsymbol{q}\leftarrow((\boldsymbol{n}'\overset{u}{\div}2)\div\boldsymbol{d})\times 2\\
\boldsymbol{r}\leftarrow\boldsymbol{n}-\boldsymbol{qd}\\
\boldsymbol{q}\leftarrow\boldsymbol{q}+(\boldsymbol{r}\overset{u}{\geqslant}\boldsymbol{d})
\end{array}
$$

9.4 无符号长除法

“长除法”（long division）的意思是用双字除以单字。在 32 位计算机上，就是 $64\overset{u}{\div}32\Rightarrow32$ 除法，当溢出（包括除以 0）时，其结果未定义（unspecified）。

某些 32 位机提供了无符号长除法指令。它功能完备，然而用处不多，因为大部分高级编程语言都只能调用 $32\overset{u}{\div}32\Rightarrow32$ 除法。因此，计算机设计者或许只提供 $32\overset{u}{\div}32$ 除法，

而且可能想要知道：实现缺失的长除法功能所用的子程序的执行时间大概是多久。本书给出两种方法来实现缺失的长除法功能。

9.4.1 用硬件实现移位并相减算法

首先要介绍的这种长除法方案是从硬件如何执行除法的角度来考虑的。有两种常见算法，叫做“恢复式除法”（restoring division）与“非恢复式除法”（nonrestoring division）［H & P，sec. A-2；EL］，二者其实都是“移位并相减”算法（shift-and-subtract algorithm）。下面演示“恢复式除法”，如果减法结果为负数，则恢复步骤（restoring step）会把除数加回去，其中的 $\boldsymbol{x}$，$\boldsymbol{y}$，$\boldsymbol{z}$ 都存于32位寄存器里。起初，双字长的被除数是 $\boldsymbol{x} \| \boldsymbol{y}$，除数是 z，需要用“1位寄存器”（single-bit register）$\boldsymbol{c}$ 来保存减法溢出情况。

```
do i←1 to 32
  c ‖ x ‖ y←2(x ‖ y)                        // Shift left one.
  c ‖ x←(c ‖ x)−(0b0 ‖ z)                   // Subtract(33 bits).
  y0←¬c                                     // Set one bit of quotient.
  if c then c ‖ x←(c ‖ x)+(0b0 ‖ z)         // Restore.
end
```

执行完毕后，商存于寄存器 $\boldsymbol{y}$，余数存于寄存器 $\boldsymbol{x}$。

在溢出时，算法得不出有意义的结果。如果用双字 $\boldsymbol{x} \| \boldsymbol{y}$ 除以0，那么商就是 $\boldsymbol{x}$ 的反码，余数就是 $\boldsymbol{y}$。尤其注意，$0 \overset{u}{\div} 0 \Rightarrow 2^{32}-1$ 余0。其他溢出情况则难以描述。

如果对于非0的除数来说能给出正确的商模 2^{32} 之后的值，而且使余数正确，那么此算法在溢出时还算有用。要做到这一点，似乎只能用97位寄存器来表示上述算法中的 $\boldsymbol{c} \| \boldsymbol{x} \| \boldsymbol{y}$，并执行64次循环。这就等于在执行 $64 \overset{u}{\div} 32 \Rightarrow 64$ 除法。减法仍然只需计算33位二进制数，不过，要实现此番改进，还需额外的硬件及执行时间，这也许不划算。

此算法难于以软件方式忠实地实现出来，因为大多数计算机都没有表示 $\boldsymbol{c} \| \boldsymbol{x}$ 所用的33位寄存器。不过，图9.2以某种程度反映了用硬件实现的“移位并相减”算法。

变量 t 用于确保比较操作的结果正确。将 x ‖ y 移位之后，需要执行33位二进制数的比较操作。若 x 首个位元是“1”（移位前），则此33位数显然大于除数（32位）。在这种情况下，x | t 这个值的每个位元都是1，于是比较操作就能得出正确结果（**true**，真）了。另一方面，若 x 首个位元是“0”，则执行32位二进制数的比较操作即可。

图9.2中的算法代码要执行321至385条基本RISC指令，具体次数取决于比较操作是否频繁为“**真**”。如果计算机支持“双字左移指令”（shift left double），则移位操作只需1条指令，而不像代码里那样需要4条。这样的话，算法执行就缩减为225至289条指令了（假设每次迭代可用两条指令来控制循环）。

以 x=0 为输入值，即可用图9.2的算法实现 $32 \overset{u}{\div} 32 \Rightarrow 32$ 除法。此时唯一能简化的地

方就是可省去变量 t，因为它总是 0。

接下来要讲以硬件实现的“非恢复式除法”算法（无符号）。基本思路是：从 33 位量 $c \| x$ 里减去除数 z 之后，就算结果为负，也不再把 z 加回去了。反之，下一轮不做减法而改做加法即可。因为，加 z（上一次循环中因减 z 而产生了偏差，所以要用加 z 来修正）、左移、再减 z，就等于直接加上 z（由 $2(u+z)-z=2u+z$ 可看出这一点）。从硬件实现的角度来看，此算法的好处是，每轮循环只有一个加法或减法操作，而加法器（adder）可能是循环中最慢的电路[⊖]。若最后的余数为负，则调整其值。（商无须调整。）

```
unsigned divlu(unsigned x, unsigned y, unsigned z) {
   // Divides (x || y) by z.
   int i;
   unsigned t;

   for (i = 1; i <= 32; i++) {
      t = (int)x >> 31;          // All 1's if x(31) = 1.
      x = (x << 1) | (y >> 31); // Shift x || y left
      y = y << 1;               // one bit.
      if ((x | t) >= z) {
         x = x - z;
         y = y + 1;
      }
   }
   return y;                    // Remainder is x.
}
```

图 9.2　用“移位并相减”算法实现无符号长除法

输入的被除数是双字量 $x \| y$，除数是 z。执行完毕时，商在寄存器 y 中，余数在寄存器 x 中。

$$
\begin{array}{ll}
\boldsymbol{c}=\mathbf{0} & \\
\text{do } i \leftarrow 1 \text{ to } 32 & \\
\quad \text{if } \boldsymbol{c}=\mathbf{0} \text{ then do} & \\
\quad\quad \boldsymbol{c} \| \boldsymbol{x} \| \boldsymbol{y} \leftarrow 2(\boldsymbol{x} \| \boldsymbol{y}) & // \text{左移 1 位} \\
\quad\quad \boldsymbol{c} \| \boldsymbol{x} \leftarrow (\boldsymbol{c} \| \boldsymbol{x})-(\mathbf{0}\boldsymbol{b}\mathbf{0} \| \boldsymbol{z}) & // \text{减去除数} \\
\quad \text{end} & \\
\quad \text{else do} & \\
\quad\quad \boldsymbol{c} \| \boldsymbol{x} \| \boldsymbol{y} \leftarrow 2(\boldsymbol{x} \| \boldsymbol{y}) & // \text{左移 1 位} \\
\quad\quad \boldsymbol{c} \| \boldsymbol{x} \leftarrow (\boldsymbol{c} \| \boldsymbol{x})+(\mathbf{0}\boldsymbol{b}\mathbf{0} \| \boldsymbol{z}) & // \text{加上除数} \\
\quad \text{end} & \\
\quad \boldsymbol{y}_0 \leftarrow \neg \boldsymbol{c} & // \text{算出商中的一个二进制位} \\
\text{end} & \\
\text{if } \boldsymbol{c}=\mathbf{1} \text{ then } \boldsymbol{x} \leftarrow \boldsymbol{x}+\boldsymbol{z} & // \text{若除数为负则调整之}
\end{array}
$$

⊖ 实际上，“恢复式除法”也可免去恢复步骤。将 33 位二进制数减法的计算结果放在另一个寄存器里，只有当其非负时，才将寄存器的值写入 x。在某些实现中，这需要多用一个寄存器，而且可能更耗时。

如将上述流程改编为32位算法，效果恐怕不太好。

801小型计算机（由IBM所制造的早期实验性RISC计算机）有“分步除法”（divide step）指令，它执行的就是上述算法循环体里面的步骤。此指令使用计算机的进位状态位（carry status bit）保存 $\boldsymbol{c}$，使用MQ（一个32位寄存器）保存 $\boldsymbol{y}$。实现此算法时要用到33位加法器/减法器。801计算机的“分步除法”指令比上述循环稍微复杂一点，因为它要执行带符号除法，而且还要检测溢出。如果使用它来实现除法子程序，那么就连续执行32次“分步除法”指令，然后调整商和余数，确保余数的正负号符合预期。

9.4.2 用短除法实现无符号长除法

将 $m=4$，$n=2$ 这个特例代入9.2节中的图9.1，即可用多字除法算法实现 $64 \overset{u}{\div} 32 \Rightarrow 32$ 除法算法。参数应该改为按值传递的全字，而非半字数组。现在的溢出情况和原来不同：当全字无法容纳商时，发生溢出。这个程序有多处可以简化。能够证明：估算出的qhat总是对的，如果除数只包含两个半字长的数位，那么估算值就是正确值。这意味着“加回”步骤可省去。图9.1的“主循环”与其中的小循环均可展开，而且还有一些小地方可以简化。

执行完这些变换之后的代码如图9.3所示。u1与u0是被除数，u1包含最高有效字（most significant word）。参数v表示除数，函数返回值是商。若调用者提供的参数r为非空指针（non-null pointer），则函数在r所指的字组中返回余数。

若程序在余数中返回了最大无符号整数，则表明除法溢出了。因为余数必须小于除数，所以有效的除法操作不可能得出这样的余数。在发生溢出时，程序返回的商等于最大无符号整数，在调用者不想要余数时，以这种方式来表明除法溢出也许比较合适。

在对u32赋值的语句中，有个奇怪的表达式：（-s≫31），这是为了令程序在执行“模32移位”的计算机上（例如Intel x86）能够正确应对s = 0时的情况。

用分步均匀的随机数实验之后，可以看出，在每次执行函数时，有大约0.38的概率会执行“again”循环体。因此可算出，如果不要余数，那么算法大约需要52条指令。这些指令中，有一条前导0计数指令，两条除法指令，6.5条乘法指令（不计与b相乘，因为那是移位指令）。如果需要余数，则多加6条（把存储r用的指令也算在内），这其中有一条乘法指令。

divlu的带符号版怎么实现呢？想逐步修改图9.3，将之改编为带符号版，这可能比较难。不过，可借用无符号版的算法来实现带符号除法：用原始参数的绝对值来调用divlu，若两个原始参数的正负号不同，则对计算结果求补。就算用绝对值最大的负数（maximum negative number）这种极端值来调用算法也没问题，因为任何带符号整数的绝对值肯定能正确表示成无符号整数。此算法列在图9.4中。

在带符号版算法中，很难编出优秀代码来检测溢出。图9.4中的算法预先判断溢出所用的方式，与对应的无符号长除法程序相同，此判断方式可以保证 $|u/v|<2^{32}$。其后，

只需确认商的符号正确或商本身是 0，即可保证不溢出了。

```
unsigned divlu(unsigned u1, unsigned u0, unsigned v,
               unsigned *r)  {
   const unsigned b = 65536; // Number base (16 bits).
   unsigned un1, un0,        // Norm. dividend LSD's.
            vn1, vn0,        // Norm. divisor digits.
            q1, q0,          // Quotient digits.
            un32, un21, un10,// Dividend digit pairs.
            rhat;            // A remainder.
   int s;                    // Shift amount for norm.

   if (u1 >= v) {            // If overflow, set rem.
      if (r != NULL)         // to an impossible value,
         *r = 0xFFFFFFFF;    // and return the largest
      return 0xFFFFFFFF;}    // possible quotient.

   s = nlz(v);               // 0 <= s <= 31.
   v = v << s;               // Normalize divisor.
   vn1 = v >> 16;            // Break divisor up into
   vn0 = v & 0xFFFF;         // two 16-bit digits.

   un32 = (u1 << s) | (u0 >> 32 - s) & (-s >> 31);
   un10 = u0 << s;           // Shift dividend left.

   un1 = un10 >> 16;         // Break right half of
   un0 = un10 & 0xFFFF;      // dividend into two digits.

   q1 = un32/vn1;            // Compute the first
   rhat = un32 - q1*vn1;     // quotient digit, q1.
again1:
   if (q1 >= b || q1*vn0 > b*rhat + un1) {
     q1 = q1 - 1;
     rhat = rhat + vn1;
     if (rhat < b) goto again1;}

   un21 = un32*b + un1 - q1*v;  // Multiply and subtract.

   q0 = un21/vn1;            // Compute the second
   rhat = un21 - q0*vn1;     // quotient digit, q0.
again2:
   if (q0 >= b || q0*vn0 > b*rhat + un0) {
     q0 = q0 - 1;
     rhat = rhat + vn1;
     if (rhat < b) goto again2;}

   if (r != NULL)            // If remainder is wanted,
      *r = (un21*b + un0 - q0*v) >> s;     // return it.
   return q1*b + q0;
}
```

图 9.3 用全字除法指令实现无符号长除法

```
int divls(int u1, unsigned u0, int v, int *r) {
   int q, uneg, vneg, diff, borrow;

   uneg = u1 >> 31;          // -1 if u < 0.
   if (uneg) {               // Compute the absolute
      u0 = -u0;              // value of the dividend u.
      borrow = (u0 != 0);
      u1 = -u1 - borrow;}

   vneg = v >> 31;           // -1 if v < 0.
   v = (v ^ vneg) - vneg;    // Absolute value of v.

   if ((unsigned)u1 >= (unsigned)v) goto overflow;
```

图 9.4 用无符号长除法实现带符号长除法

```
    q = divlu(u1, u0, v, (unsigned *)r);

    diff = uneg ^ vneg;          // Negate q if signs of
    q = (q ^ diff) - diff;       // u and v differed.
    if (uneg && r != NULL)
       *r = -*r;

    if ((diff ^ q) < 0 && q != 0) {  // If overflow,
overflow:                        // set remainder
       if (r != NULL)            // to an impossible value,
          *r = 0x80000000;       // and return the largest
       q = 0x80000000;}          // possible neg. quotient.
    return q;
}
```

图 9.4 （续）

9.5 用长除法实现双字除法

本节思考如何由 64÷32⇒32 除法算出 64÷64⇒64 除法，带符号与无符号两种情况都要考虑。下面要讲的算法最适合那些支持长除法（64÷32）指令的计算机，至少无符号版本是如此。如果计算机有前导0计数指令，那么也有助于实现这些算法。计算机可能有32位或64位寄存器，然而我们假定：如果是32位寄存器，那么使用64位操作数（C语言的“long long”数据类型）的加法及移位操作将由编译器实现。

这些函数在GNU C中叫做“__udivdi3”与“__divdi3”，本书也使用类似名称。

9.5.1 无符号双字除法

图 9.5 列出了此操作的程序代码。

此代码分别处理这三种情况：(1) 仅用计算机的无符号长除法指令（DIVU）即可实现的除法；(2) 不符合情况（1）但除数为32位量的除法；(3) 除数用32个二进制位容纳不下的除法。上述代码能正确应对情况（1）、(2)，这不难理解。思考一下如何用竖式做长除法，就能明白上述算法是如何处理情况（2）的了。

情况（3）需要证明，因为在某些时候此算法看上去似乎不太可能正常运作。请注意，情况（3）仅需执行一次DIVU指令，然而它需要前导0计数指令与乘法操作。

为了证明，我们需要了解以下基本知识（对整数变量成立）：

$$\lfloor \lfloor a/b \rfloor /d \rfloor = \lfloor a/(bd) \rfloor \tag{2}$$

$$b\lfloor a/b \rfloor = a - \text{rem}(a,b) \tag{3}$$

由算法相关部分的第一行代码可知（假设 $v \neq 0$）：

$$0 \leqslant n \leqslant 31$$

在计算 v_1 时，左移操作显然不会溢出。因此

$$v_1 = \lfloor v/2^{32-n} \rfloor$$

$$u_1 = \lfloor u/2 \rfloor$$

```
unsigned long long udivdi3(unsigned long long u,
                           unsigned long long v) {

   unsigned long long u0, u1, v1, q0, q1, k, n;

   if (v >> 32 == 0) {            // If v < 2**32:
      if (u >> 32 < v)            // If u/v cannot overflow,
         return DIVU(u, v)        // just do one division.
            & 0xFFFFFFFF;
      else {                      // If u/v would overflow:
         u1 = u >> 32;            // Break u up into two
         u0 = u & 0xFFFFFFFF;     // halves.
         q1 = DIVU(u1, v)         // First quotient digit.
            & 0xFFFFFFFF;
         k = u1 - q1*v;           // First remainder, < v.
         q0 = DIVU((k << 32) + u0, v) // 2nd quot. digit.
            & 0xFFFFFFFF;
         return (q1 << 32) + q0;
      }
   }
                                  // Here v >= 2**32.
   n = nlz64(v);                  // 0 <= n <= 31.
   v1 = (v << n) >> 32;           // Normalize the divisor
                                  // so its MSB is 1.
   u1 = u >> 1;                   // To ensure no overflow.
   q1 = DIVU(u1, v1)              // Get quotient from
       & 0xFFFFFFFF;              // divide unsigned insn.
   q0 = (q1 << n) >> 31;          // Undo normalization and
                                  // division of u by 2.
   if (q0 != 0)                   // Make q0 correct or
      q0 = q0 - 1;                // too small by 1.
   if ((u - q0*v) >= v)
      q0 = q0 + 1;                // Now q0 is correct.
   return q0;
}
```

图 9.5 用长除法实现无符号双字除法

在计算 q_1 时，因为 u_1 和 v_1 都在 DIVU 指令所允许的操作数范围内，所以除法不会溢出。故而

$$q_1 = \lfloor u_1 / v_1 \rfloor$$

首次计算 q_0 时，因为 $q_1 < 2^{32}$（由于 u_1 最大值是 $2^{63}-1$，而 v_1 的最小值是 2^{31}），所以左移操作不会溢出。故而

$$q_0 = \lfloor q_1 / 2^{31-n} \rfloor$$

现在来到整个证明的主体部分了，我们需要求证

$$\lfloor u/v \rfloor \leqslant q_0 \leqslant \lfloor u/v \rfloor + 1$$

也就是说，首次算出的 q_0 值要么是正确答案，要么比正确答案多 1。

运用两次等式（2），可得

$$\begin{aligned} q_0 &= \left\lfloor \frac{u}{2^{32-n} v_1} \right\rfloor \\ &= \left\lfloor \frac{u}{2^{32-n} \left\lfloor \dfrac{v}{2^{32-n}} \right\rfloor} \right\rfloor \end{aligned}$$

运用等式（3），可得

$$q_0=\left\lfloor\frac{u}{v-\mathrm{rem}(v,2^{32-n})}\right\rfloor$$

运用代数技巧将上式改写为“u/v+某值”的形式：

$$q_0=\left\lfloor\frac{u}{v}+\frac{u\ \mathrm{rem}(v,2^{32-n})}{v(v-\mathrm{rem}(v,2^{32-n}))}\right\rfloor$$

这就具备了下列形式：

$$\left\lfloor\frac{u}{v}+\delta\right\rfloor$$

我们需证明 $\delta<1$。

当 $\mathrm{rem}(v,2^{32-n})$ 尽可能大而 v 又尽可能小时，δ 最大。$\mathrm{rem}(v,2^{32-n})$ 最大值是 $2^{32-n}-1$。因为 n 取决于 v，而 $v\geqslant 2^{63-n}$，因此，令余数最大的最小 v 值是

$$2^{63-n}+2^{32-n}-1$$

因此

$$\begin{aligned}\delta&\leqslant\frac{u(2^{32-n}-1)}{(2^{63-n}+2^{32-n}-1)2^{63-n}}\\&<\frac{u(2^{32-n}-1)}{(2^{63-n})^2}\end{aligned}$$

经检验可知，当 n 值在 0 至 31 之间时，

$$\delta<\frac{u}{2^{64}}$$

因为 u 最大是 $2^{64}-1$，所以 $\delta<1$。因为 $q_0=\lfloor u/v+\delta\rfloor$ 且 $\delta<1$（显然 $\delta>0$），所以

$$\left\lfloor\frac{u}{v}\right\rfloor\leqslant q_0\leqslant\left\lfloor\frac{u}{v}\right\rfloor+1$$

为了在必要时把结果减 1，我们可能需要如下代码：

```
if (u < q0*v) q0 = q0 - 1;
```

（也就是说，如果余数 $u-q_0v$ 是负值，则把 q_0 减 1。）然而，这么做不太行，因为 q_0v 可能溢出（例如，当 $u=2^{64}-1$ 而 $v=2^{32}+3$ 时）。于是，先把 q_0 减 1，这样的话，它要么正确，要么比正确值小 1。然后再计算 q_0v，此时就不会溢出了。注意，$q_0=0$ 时不能减 1（因为如果 $q_0=0$，那么它已经是正确的商了）。

于是，最后的修正步骤就是：

```
if ((u - q0*v) >= v) q0 = q0 + 1;
```

为了理解这样能算出正确结果，我们首先注意到 q_0v 已经不会再溢出了。很容易证明下列关系式：

$$0\leqslant u-q_0v<2v$$

如果 v 非常大（$\geqslant 2^{63}$），那么减法会不会在其结果大于 v 时溢出呢？不会，因为 $u<2^{64}$ 且 $q_0v\geqslant 0$。

顺便说说，在某些计算机中，把下面这两行代码改为其他写法效果会更好：

```
if (q0 != 0)          // Make q0 correct or
   q0 = q0 - 1        // too small by 1.
```

一种替代方案是：

```
if (q0 == 0) return 0;
```

另外一种方法是在这一段程序码或整个函数前加一句：

```
if (u < v) return 0; // Avoid a problem later.
```

如果分支指令开销不大，那么上面这些替代方案都很好。若计算机的比较指令能产生 0/1 整数值，并将其放在通用寄存器中，则图 9.5 中的代码效果不错，因为此时编译器可以把这两行改写成与下列代码等效的指令：

```
q0 = q0 - (q0 != 0);
```

（要是编译器没有按此优化，那么开发者可以自己将代码改为这种形式）。在此类计算机上，只需一条比较指令和一条减法指令即可实现。

9.5.2 带符号双字除法

在带符号的情况下做双字除法，最好的办法似乎就是：先把操作数取绝对值，用 udivdi3 函数做除法，若原操作数的正负号相异，则把商设为其相反数。如果计算机有带符号长除法指令（此处将之称为 DIVS），那么，把能用 DIVS 求值的情况单列出来而不再调用 udivdi3，效果也许更好。这么做表明此种情况较为常见。图 9.6 列出了这样的函数。

```
#define llabs(x) \
   ({unsigned long long t = (x) >> 63; ((x) ^ t) - t;})

   long long divdi3(long long u, long long v) {

   unsigned long long au, av;
   long long q, t;

   au = llabs(u);
   av = llabs(v);
   if (av >> 31 == 0) {          // If |v| < 2**31 and
      if (au < av << 31) {       // |u|/|v| cannot
         q = DIVS(u, v);         // overflow, use DIVS.
         return (q << 32) >> 32;
      }
   }
   q = au/av;                    // Invoke udivdi3.
   t = (u ^ v) >> 63;            // If u, v have different
   return (q ^ t) - t;           // signs, negate q.
}
```

图 9.6 用无符号双字除法实现带符号双字除法

图 9.6 的代码中出现了“#define”，它利用 GCC 编译器的一项功能，以圆括号内的“复合语句”（compound statement）来构建表达式，大部分 C 语言编译器都没有此功能。

某些编译器也许有名为 llabs(x) 的内置函数（built-in function）。

代码不能精确判断出 v 是否在可以使用 DIVS 的范围内，若 $v=-2^{31}$，则判断错误。如果非要在这种情况下使用 DIVS 指令来计算的话，那么可以用下列代码

```
if ((v<<32) >>32 == v) { // If v is in range and
```

取代图 9.6 中的第 3 行可执行代码（修改之后会多花 1 条指令）。与之类似，算法用比较简单的代码来测试 $|u|/|v|$ 是否溢出，这种方法也会误判一些"特殊情况"（corner case）。2.13 节曾讲了一种判断带符号除法是否溢出的方案，本算法采取的方法就等于将那个测试方案中的 δ 视为 0。

9.6 习题

1. 证明 $\lfloor x \rfloor = -\lceil x \rceil$ 对任意实数 x 都成立。
2. 在一台使用"向 0 取整式除法"（truncating division，截取除法）来执行除法及求余指令的基本 RISC 计算机上，编写无分支代码，算出"余数非负式除法"（modulus division，"模"除法）的商和余数。
3. 与上题类似，在一台使用"向 0 取整式除法"来执行除法及求余指令的基本 RISC 计算机上，编写无分支代码，算出"向下取整式除法"（floor division，地板除法）的商和余数。
4. 已知两个无符号整数 $\boldsymbol{n}$，$\boldsymbol{d}$，$0 \leqslant n \leqslant 2^{32}-1$ 且 $1 \leqslant d \leqslant 2^{32}-1$，如何求出 $\lceil \boldsymbol{n}/\boldsymbol{d} \rceil$？假定计算机具备可算出 $\lfloor \boldsymbol{n}/\boldsymbol{d} \rfloor$ 的无符号除法指令。
5. 定理 D3 说：对实数 x 与整数 d，$\lfloor \lfloor x \rfloor / d \rfloor = \lfloor x/d \rfloor$ 成立。请证明比该定理更通用的命题：若 $f(x)$ 同时满足（a）连续、（b）单调递增、（c）当 $f(x)$ 是整数时 x 也是整数，则 $\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$［GKP］。

第 10 章

除数为常量的整数除法

在许多计算机中，除法非常耗时，应尽量避免。如果执行除法所耗时间是简单加法指令的至少 20 倍，那并不算稀奇，而且即便除法的操作数较小，通常也会很耗时。本章给出一些在除数为常量时免于执行除法指令的方法。

10.1 除数为 2 的已知次幂的带符号除法

许多人似乎都误认为：把某数带符号右移 k 位，就等于用通常的“向 0 取整式除法”(truncating form of division) 将它除以 2^k[GLS2]。实际情况要比预想的复杂一点。下列代码可计算 $\boldsymbol{q}=\boldsymbol{n}\div\boldsymbol{2}^k$ 的值，其中 $1\leqslant k\leqslant 31$ [Hop]。

```
shrsi t,n,k-1        Form the integer
shri  t,t,32-k       2**k - 1 if n < 0, else 0.
add   t,n,t          Add it to n,
shrsi q,t,k          and shift right (signed).
```

此代码无分支。如果遇到除数是 $2(k=1)$ 这种常见情况，那么就可简化成 3 条指令。然而，此算法的效果依赖于计算机能否在短时间内对一个大数执行移位操作。$k=31$ 的情况没多大意义，因为计算机无法表示 2^{31} 这个数。不过，在此种情况下此算法的结果也正确（当 $n=-2^{31}$ 时，可算出 $q=-1$，当 n 为其余值时，可算出 $q=0$）。

如果要除以 -2^k，那么可在上述算法后加一条取相反数 (negate) 指令。似乎没有更好的办法了。

还有种更直观的代码能计算除以 2^k：

```
      bge   n,label       Branch if n >= 0.
      addi  n,n,2**k-1    Add 2**k - 1 to n,
label shrsi n,n,k         and shift right (signed).
```

如果计算机的移位操作慢而分支操作快，那么这种办法更好。

PowerPC 计算机有种非常规手段能够迅速除以 2 的幂 [GGS]。如果被右移的数为负，且至少有 1 个值为“1”的位元移出右边界了，那么带符号右移操作就会设置计算机的进位标志。此计算机还有一条名为 addze 的指令，可把进位标志加到某个寄存器上。这

样的话，除以 2 的正整数幂就可用两条指令实现：

```
shrsi q,n,k
addze q,q
```

如果单用 shrsi 指令右移 k 位，那么其效果就碰巧等于用“余数非负式除法”或“向下取整式除法”将带符号整数除以 2^k。由此可见，对计算机和高级程序语言来说，这两种除法也许比“向 0 取整式除法”更好。也就是说，和“向 0 取整式除法”相比，“余数非负式除法”及“向下取整式除法”与 shrsi 更搭调，因为编译器可把表达式 $n/2$ 转译成一条 shrsi 指令。而且，执行完 shrsi 之后再执行 neg（negate，取相反数）指令，也就等于用“余数非负式除法”除以 -2^k，这也暗示着“余数非负式除法”最好。（其实主要是个审美问题，没多少实际意义，因为用负常数做除数必定相当罕见。）

10.2 求与 2 的已知次幂相除的带符号余数

如果想一并求出 $\boldsymbol{n} \div \mathbf{2}^k$ 的商和余数，那么最简单的方法就是用 $\boldsymbol{r}=\boldsymbol{n}-\boldsymbol{q} * \mathbf{2}^k$ 算出余数 $\boldsymbol{r}$。求出商数 $\boldsymbol{q}$ 后，只需再用两条指令即可：

```
shli  r,q,k
sub   r,n,r
```

要是只想求余数，那么似乎需要四五条指令。一种方法是，用上节讲的那 4 条指令计算除数为 2^k 的带符号除法，然后用刚才讲的那两条求出余数。这样的话，相邻的那两条移位指令就能合并成一个“与”操作，于是共需 5 条指令（若 $k=1$，则只需 4 条）。

```
shrsi t,n,k-1        Form the integer
shri  t,t,32-k       2**k - 1 if n < 0, else 0.
add   t,n,t          Add it to n,
andi  t,t,-2**k      clear rightmost k bits,
sub   r,n,t          and subtract it from n.
```

另一种方法是基于下列公式：

$$\operatorname{rem}(n,2^k)=\begin{cases} n \,\&(2^k-1), n\geqslant 0, \\ -((-n)\,\&(2^k-1)), n<0 \end{cases}$$

首先计算 $\boldsymbol{t} \leftarrow \boldsymbol{n} \overset{s}{\gg} 31$，然后计算下式（需要 5 条指令）：

$$\boldsymbol{r} \leftarrow ((\operatorname{abs}(\boldsymbol{n})\,\&\,(\mathbf{2}^k-\mathbf{1}))\oplus \boldsymbol{t})-\boldsymbol{t}$$

如果 $k=1$，那么因为 $(-\boldsymbol{n})\,\&\,\mathbf{1}=\boldsymbol{n}\,\&\,\mathbf{1}$，所以可化简为下式（需要 4 条指令）：

$$\boldsymbol{r} \leftarrow ((\boldsymbol{n}\,\&\,\mathbf{1})\oplus \boldsymbol{t})-\boldsymbol{t}$$

如果计算机没有求绝对值指令，那么此方法在 $k>1$ 时不算太好（在这种情况下，求余数需要 6 条指令）。

此外还有个基于下式的方法：

$$\operatorname{rem}(n,2^k)=\begin{cases} n \,\&(2^k-1), n\geqslant 0 \\ ((n+2^k-1)\,\&(2^k-1))-(2^k-1), n<0 \end{cases}$$

该式子可归结为如下算法：

$$t \leftarrow (n \overset{s}{\gg} k-1) \overset{u}{\gg} 32-k$$
$$r \leftarrow ((n+t) \& (2^k-1))-t$$

（$k>1$ 时需 5 条指令，$k=1$ 则需 4 条。）

上面所有方法在 $1 \leqslant k \leqslant 31$ 时都有效。

顺便说说，如果没有带符号右移指令，那么用下面的方法也能构建一个值，令其在 $n<0$ 时为 2^k-1，在 $n \geqslant 0$ 时为 0：

$$t_1 \leftarrow n \overset{u}{\gg} 31$$
$$r \leftarrow (t_1 \ll k)-t_1$$

这比用带符号右移来实现多了 1 条指令。

10.3 在除数不是 2 的幂时求带符号除法及余数

基本技巧是：和一个与除数 d 的倒数相似之数（大约是 $2^{32}/d$）做乘法，然后提取积的左 32 位。然而细节却比较复杂，尤其在除数为 7 这样的值时更是如此。

首先思考更具体的例子。通过这些例子，可以了解通用算法的代码是什么样子。我们定义下列寄存器：

n ——存放输入的整数（分子）
M ——用于载入“奇妙数字”（magic number）
t ——临时寄存器
q ——将存放商
r ——将存放余数

10.3.1 除以 3

```
li    M,0x55555556  Load magic number, (2**32+2)/3.
mulhs q,M,n         q = floor(M*n/2**32).

shri  t,n,31        Add 1 to q if
add   q,q,t         n is negative.

muli  t,q,3         Compute remainder from
sub   r,n,t         r = n - q*3.
```

证明：因为两个 32 位整数之积可用 64 位表示，而 mulhs 取这 64 位里权重高的 32 位，所以“求带符号乘法乘积的高权重部分”（multiply high signed operation，mulhs）这一操作不可能溢出。不论这个 64 位乘积是正是负，此操作等于把它除以 2^{32}，并向下取整。因此，在 $n \geqslant 0$ 时，上述算法计算的就是

$$q=\left\lfloor \frac{2^{32}+2}{3} \frac{n}{2^{32}} \right\rfloor=\left\lfloor \frac{n}{3}+\frac{2n}{3 \cdot 2^{32}} \right\rfloor$$

由于用 32 位可表示出来的最大整数是 $2^{31}-1$，所以 $n<2^{31}$。这样的话，“误差项”

$2n/(3\cdot 2^{32})$ 就小于 1/3（而且非负），于是由定理 D4（9.1 节）可得 $q=\lfloor n/3\rfloor$，根据 9.1 节等式（1）可知，这就是预期结果。

当 $n<0$ 时，算法会给商加 1。于是代码算的就是（在推理过程中，运用了定理 D2）：

$$q=\left\lfloor\frac{2^{32}+2}{3}\frac{n}{2^{32}}\right\rfloor+1=\left\lfloor\frac{2^{32}n+2n+3\cdot 2^{32}}{3\cdot 2^{32}}\right\rfloor=\left\lceil\frac{2^{32}n+2n+1}{3\cdot 2^{32}}\right\rceil$$

于是可得

$$q=\left\lceil\frac{n}{3}+\frac{2n+1}{3\cdot 2^{32}}\right\rceil$$

当 $-2^{31}\leqslant n\leqslant -1$ 时，

$$-\frac{1}{3}+\frac{1}{3\cdot 2^{32}}\leqslant\frac{2n+1}{3\cdot 2^{32}}\leqslant-\frac{1}{3\cdot 2^{32}}$$

因为误差项不是正值，而且大于 $-1/3$，所以根据定理 D4 可得 $q=\lfloor n/3\rfloor$。由 9.1 节等式（1）可知，这正是想要的答案。

综上可知，商是正确的。因为余数必定满足

$$n=qd+r$$

而 q 乘以 3 不可能溢出（因为 $-2^{31}/3\leqslant q\leqslant(2^{31}-1)/3$），减法操作也不可能溢出（因为结果必定在 -2 至 $+2$ 的范围内），所以很容易就能证明余数也正确。 ■

与常量相乘（multiply immediate，muli）这一指令可改写为两次加法，或一次移位与一次加法。只要其中一种改法能缩短执行时间，那么就值得修改。

在当前大多数 RISC 架构的计算机中，用上述办法求商大约要 9 或 10 个周期，而除法指令也许要花 20 个周期或更长时间。

10.3.2 除以 5

如果要除以 5，那么可以使用除以 3 的方法，只是乘数要换为 $(2^{32}+4)/5$。但这种方法的误差项太大，当 $n\geqslant 2^{30}$ 时，计算结果有 1/5 的几率会与正确答案相差 1。不过，只要把乘数改为 $(2^{33}+3)/5$ 并加上一条带符号右移指令就好了。此算法的代码为：

```
li    M,0x66666667  Load magic number, (2**33+3)/5.
mulhs q,M,n         q = floor(M*n/2**32).
shrsi q,q,1
shri  t,n,31        Add 1 to q if
add   q,q,t         n is negative.

muli  t,q,5         Compute remainder from
sub   r,n,t         r = n - q*5.
```

证明： 算法用 mulhs 获取 64 位乘积的左 32 位，然后将此值带符号（或曰"算术"）右移 1 位。这就等于把 64 位积除以 2^{33} 并向下取整。因此，当 $n\geqslant 0$ 时，代码计算的是

$$q=\left\lfloor\frac{2^{33}+3}{5}\frac{n}{2^{33}}\right\rfloor=\left\lfloor\frac{n}{5}+\frac{3n}{5\cdot 2^{33}}\right\rfloor$$

当 $0\leqslant n<2^{31}$ 时，误差项 $3n/5\cdot 2^{33}$ 非负且小于 1/5，根据定理 D4 可知，$q=\lfloor n/5\rfloor$。

当 $n<0$ 时，上述代码计算的是

$$q=\left\lfloor \frac{2^{33}+3}{5}\frac{n}{2^{33}} \right\rfloor+1=\left\lceil \frac{n}{5}+\frac{3n+1}{5\cdot 2^{33}} \right\rceil$$

由于误差项不是正值且大于 $-1/5$，所以 $q=\lceil n/5 \rceil$。

和除以 3 时的情况相同，也可证明余数正确。

其中与常量相乘这一操作，可改为将 q 左移两位再和其自身相加。■

10.3.3 除以 7

除以 7 时会碰到新问题。用 $(2^{32}+3)/7$ 与 $(2^{33}+6)/7$ 作乘数，误差项都太大了。用 $(2^{34}+5)/7$ 作乘数可以，但是它本身太大了，无法保存在 32 位带符号字组中。我们可以和 $(2^{34}+5)/7-2^{32}$（这是个负数）相乘，然后用加法指令来修正乘积，这样也就等于和 $(2^{34}+5)/7$ 这个大数相乘了。此算法的代码是：

```
li    M,0x92492493  Magic num, (2**34+5)/7 - 2**32.
mulhs q,M,n         q = floor(M*n/2**32).
add   q,q,n         q = floor(M*n/2**32) + n.
shrsi q,q,2         q = floor(q/4).
shri  t,n,31        Add 1 to q if
add   q,q,t         n is negative.

muli  t,q,7         Compute remainder from
sub   r,n,t         r = n - q*7.
```

证明：注意到代码中的 “add q，q，n” 指令不会溢出，这一点很重要。因为乘法指令将 n 与负数相乘，所以导致 q 与 n 正负号相反。因此，这个“计算机算术”加法就与普通的实数加法一样了。于是，在 $n\geqslant 0$ 时，上述代码计算的是（推理过程中用到了定理 D3 的推论）：

$$\begin{aligned} q &= \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7}-2^{32} \right)\frac{n}{2^{32}} \right\rfloor + n \right)/4 \right\rfloor = \left\lfloor \left\lfloor \frac{2^{34}n+5n-7\cdot 2^{32}n+7\cdot 2^{32}n}{7\cdot 2^{32}} \right\rfloor /4 \right\rfloor \\ &= \left\lfloor \frac{n}{7}+\frac{5n}{7\cdot 2^{34}} \right\rfloor \end{aligned}$$

当 $0\leqslant n\leqslant 2^{31}$ 时，误差项 $5n/\ 7\cdot 2^{34}$ 非负且小于 $1/7$，所以 $q=\lfloor n/7 \rfloor$。

当 $n<0$ 时，上述代码计算的是：

$$q=\left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7}-2^{32} \right)\frac{n}{2^{32}} \right\rfloor + n \right)/4 \right\rfloor+1=\left\lceil \frac{n}{7}+\frac{5n+1}{7\cdot 2^{34}} \right\rceil$$

其中的误差项不是正数且小于 $-1/7$，所以 $q=\lceil n/7 \rceil$。■

与常量相乘这一操作，可以改为将 q 左移 3 位然后再减 q。

10.4 除数大于等于 2 的带符号除法

看到这里，读者也许要问，计算除数为其他常量的除法时，会不会又遭遇新问题呢？本节将告诉大家：不会的，上一节给出的 3 个例子已经把（当 $d\geqslant 2$ 时）所有可能出问题

的情况演示完了。

阅读时请小心，证明过程中的某些步骤有点复杂，下文用 W 作为字长，这样更通用些。

给定字长 W 与除数 d，并且知道 $W \geqslant 3$、$2 \leqslant d \leqslant 2^{W-1}$，现在需要寻找最小整数 m 与整数 p，使得：

$$当\ 0 \leqslant n < 2^{W-1}\ 时，\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \tag{1a}$$

$$当\ -2^{W-1} \leqslant n \leqslant -1\ 时，\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lfloor \frac{n}{d} \right\rfloor \tag{1b}$$

其中 $0 \leqslant m < 2^W$ 且 $p \geqslant W$。

令整数 m 最小的原因在于：乘数如果小，那么移位量也许就比较小（甚至可能是0），或者说，这么做可以产生"除以5"所用的那种简单代码，而不是"除以7"时所用的那种复杂代码。必须规定 $m \leqslant 2^W - 1$，这样的话，代码所需的指令数就不会超过"除以7"那个例子了（也就是说，我们可以像"除以7"时那样，通过插入加法指令来处理乘数在 2^{W-1} 与 $2^W - 1$ 之间的情况，但是不想处理比这更大的乘数了）。还必须规定 $p \geqslant W$，因为生成的算法代码要把乘积 mn 的左半边提取出来，这等于要把乘积右移 W 位。因此，总右移量必须是 W 位或更多位才行。

乘数 m 与上一节说的"神奇数字" M 之间有区别。神奇数字是乘法指令中用到的那个值，它由下式算出：

$$M = \begin{cases} m & 若\ 0 \leqslant m < 2^{W-1} \\ m - 2^W & 若\ 2^{W-1} \leqslant m < 2^W \end{cases}$$

因为（1b）在 $n = -d$ 时必定成立，所以可知 $\lfloor -md/2^p \rfloor + 1 = -1$ 必定成立，这就意味着

$$\frac{md}{2^p} > 1 \tag{2}$$

设 n_c 为满足 $\mathrm{rem}(n_c, d) = d - 1$ 的最大正值。这样的 n_c 肯定存在，因为 $n_c = d - 1$ 这个取值满足条件。n_c 的值可以这么来算：$n_c = \lfloor 2^{W-1}/d \rfloor d - 1 = 2^{W-1} - \mathrm{rem}(2^{W-1}, d) - 1$。在 n 的取值范围中最大的 d 个数中，必有一个是 n_c，所以：

$$2^{W-1} - d \leqslant n_c \leqslant 2^{W-1} - 1 \tag{3a}$$

而且，显然可知

$$n_c \geqslant d - 1 \tag{3b}$$

因为（1a）在 $n = n_c$ 时成立，所以

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d}$$

或者说

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}$$

将上式与（2）结合，可得

$$\frac{2^p}{d}<m<\frac{2^p}{d}\frac{n_c+1}{n_c} \tag{4}$$

由于 m 是满足（4）的最小整数，所以它是比 $2^p/d$ 大的下一个整数，也就是说

$$\boxed{m=\frac{2^p+d-\mathrm{rem}(2^p,d)}{d}} \tag{5}$$

将上式和（4）的右半边结合，并化简，可得：

$$\boxed{2^p>n_c(d-\mathrm{rem}(2^p,d))} \tag{6}$$

10.4.1 算法

因此，想编写算法找到“奇妙数字”M 并根据 d 找出移位量 s，先要算出 n_c，然后根据关系式（6）连续尝试大数值以解出 p。如果 $p<W$，则设定 $p=W$（根据下面要讲的定理可知这个 p 值也满足关系式（6））。如果找到符合 $p\geqslant W$ 且满足（6）式的最小 p 值，那么根据（5）式计算 m。因为已经找到了合格的最小 p 值，而且根据（4）显然可知 p 的值越小则 m 的值也越小，所以这么算出来的 m 就是它可能取到的最小值。最后可算出 $s=p-W$，而把 m 表示为带符号数，也就得到了 M 的值（mulhs 指令就是这么解释 m 的）。

下面证明为何能够将 p 强制上调为 W：

定理 DC1　如果（6）式对某个 p 值成立，则它对所有更大的 p 值也成立。

证明： 假设当 $p=p_0$ 时(6)式成立。将(6)式左右两端乘以 2，可得

$$2^{p_0+1}>n_c(2d-2\mathrm{rem}(2^{p_0},d))$$

根据定理 D5 可知，$\mathrm{rem}(2^{p_0+1},d)\geqslant 2\mathrm{rem}(2^{p_0},d)-d$，将此式与上式结合起来，可得

$$2^{p_0+1}>n_c(2d-(\mathrm{rem}(2^{p_0+1},d)+d))\text{，或者说}$$

$$2^{p_0+1}>n_c(d-\mathrm{rem}(2^{p_0+1},d))$$

因此，（6）式对 $p=p_0+1$ 也成立，因此它对大于 p_0 的所有 p 值都成立。 ■

于是可用二分查找来求解（6）式，当然，因为通常 d 值较小，而 d 值小时 p 值也小，所以，用简单的线性查找（从 $p=W$ 开始找）也许更好。

10.4.2 算法可行性证明

必须证明（6）式总有解，且根据其值算出来的 m 符合 $0\leqslant m<2^W$。（没必要证明 $p\geqslant W$，因为这是强制规定。）

只要找到 p 的上界，就可证明（6）式必然有解。从更加通用的角度考虑，不强制要求 p 大于等于 W，在此前提下也找出 p 的下界。为了寻找 p 的上下界，我们注意到：对任何正整数 x 来说，都能找到一个 2 的整数幂，它大于 x 而小于 $2x$。因此，根据（6）式

可得：

$$n_c(d-\text{rem}(2^p,d))<2^p\leqslant 2n_c((d-\text{rem}(2^p,d))$$

因为 $0\leqslant \text{rem}(2^p,d)\leqslant d-1$，所以

$$n_c+1\leqslant 2^p\leqslant 2n_cd \tag{7}$$

由（3a）与（3b）可知，$n_c\geqslant \max(2^{W-1}-d,d-1)$。由 $f_1(d)=2^{W-1}-d$ 及 $f_2(d)=d-1$ 这两个函数所确定的两条直线会在 $d=(2^{W-1}+1)/2$ 时相交。因此，$n_c\geqslant(2^{W-1}-1)/2$。又因为 n_c 是整数，所以 $n_c\geqslant 2^{W-2}$。由于 n_c，$d\leqslant 2^{W-1}-1$，所以（7）式就变成：

$$2^{W-2}+1\leqslant 2^p\leqslant 2(2^{W-1}-1)^2$$

或者说

$$W-1\leqslant p\leqslant 2W-2 \tag{8}$$

当 $p=W-1$ 时 p 能够达到下界（例如 $W=32$ 且 $d=3$），不过在该种情况下我们会设定 $p=W$。

如果不强迫 p 与 W 相等，那么根据（4）与（7），

$$\frac{n_c+1}{d}<m<\frac{2n_cd}{d}\frac{n_c+1}{n_c}$$

根据（3b）可得

$$\frac{d-1+1}{d}<m<2(n_c+1)$$

根据（3a）可知 $n_c\leqslant 2^{W-1}-1$，所以

$$2\leqslant m\leqslant 2^W-1$$

若强令 p 等于 W，则根据（4）可得：

$$\frac{2^W}{d}<m<\frac{2^W}{d}\frac{n_c+1}{n_c}$$

由于 $2\leqslant d\leqslant 2^{W-1}-1$ 且 $n_c\geqslant 2^{W-2}$，所以

$$\frac{2^W}{2^{W-1}-1}<m<\frac{2^W}{2}\frac{2^{W-2}+1}{2^{W-2}}$$

$$3\leqslant m\leqslant 2^{W-1}+1$$

因此，不论哪种情况，从“除以7”的那个例子所演示的代码方案来看，m 均在其范围内。

10.4.3 证明乘积正确

现在要证明：若 p 与 m 分别根据式（6）、（5）算出，则等式（1a）与（1b）成立。

等式（5）与不等式（6）显然保证了不等式（4）成立。（在强令 p 等于 W 时，由定理DC1可知，（6）依然成立。）下面分别讨论 n 位于下列5个区间内的情况：

$$0\leqslant n\leqslant n_c$$

$$n_c+1 \leqslant n \leqslant n_c+d-1$$
$$-n_c \leqslant n \leqslant -1$$
$$-n_c-d+1 \leqslant n \leqslant -n_c-1$$
$$n=-n_c-d$$

由（4）可知，因为 m 是整数，所以

$$\frac{2^p}{d}<m \leqslant \frac{2^p(n_c+1)-1}{dn_c}$$

为不等式乘以 $n/2^p$，在 $n \geqslant 0$ 时，它就变成：

$$\frac{n}{d} \leqslant \frac{mn}{2^p} \leqslant \frac{2^p n(n_c+1)-n}{2^p dn_c}, \text{因此,}$$

$$\left\lfloor \frac{n}{d} \right\rfloor \leqslant \left\lfloor \frac{mn}{2^p} \right\rfloor \leqslant \left\lfloor \frac{n}{d}+\frac{(2^p-1)n}{2^p dn_c} \right\rfloor$$

在 $0 \leqslant n \leqslant n_c$ 时，$0 \leqslant (2^p-1)n/(2^p dn_c)<1/d$，所以根据定理 D4，

$$\left\lfloor \frac{n}{d}+\frac{(2^p-1)n}{2^p dn_c} \right\rfloor=\left\lfloor \frac{n}{d} \right\rfloor$$

因此（1a）在此情况（$0 \leqslant n \leqslant n_c$）下成立。

如果 $n>n_c$，则 n 位于下列范围内：

$$n_c+1 \leqslant n \leqslant n_c+d-1 \tag{9}$$

因为要在 n 的取值范围中最大的那些数里选取满足 $\mathrm{rem}(n_c, d)=d-1$ 的 n_c，所以绝对不会发生 $n \geqslant n_c+d$ 的情况（或者根据（3a）可看出，若 $n \geqslant n_c+d$，则 $n \geqslant 2^{W-1}$）。由（4）式可得，当 $n \geqslant 0$ 时：

$$\frac{n}{d}<\frac{mn}{2^p}<\frac{n}{d}\frac{n_c+1}{n_c}$$

根据初等代数知识，可将上式写成

$$\frac{n}{d}<\frac{mn}{2^p}<\frac{n_c+1}{d}+\frac{(n-n_c)(n_c+1)}{dn_c} \tag{10}$$

根据（9）式可知 $1 \leqslant n-n_c \leqslant d-1$，所以

$$0<\frac{(n-n_c)(n_c+1)}{dn_c} \leqslant \frac{d-1}{d}\frac{n_c+1}{n_c}$$

根据（3b）可知 $n_c \geqslant d-1$，而当 n_c 取最小值时 $(n_c+1)/n_c$ 最大，所以：

$$0<\frac{(n-n_c)(n_c+1)}{dn_c} \leqslant \frac{d-1}{d}\frac{d-1+1}{d-1}=1$$

在（10）中，$(n_c+1)/d$ 这一项是个整数，而 $(n-n_c)(n_c+1)/dn_c$ 这一项小于等于 1。因此，（10）就变成

$$\left\lfloor \frac{n}{d} \right\rfloor \leqslant \left\lfloor \frac{mn}{2^p} \right\rfloor \leqslant \frac{n_c+1}{d}$$

对位于（9）式范围内的所有 n 值来说，$\lfloor n/d \rfloor=(n_c+1)/d$。因此，在这种情况下（$n_c+$

$1\leqslant n\leqslant n_c+d-1$)，等式（1a）成立。

现在证明 $n<0$ 的情况。由于 m 是整数，所以根据（4）式可得

$$\frac{2^p+1}{d}\leqslant m<\frac{2^p}{d}\frac{n_c+1}{n_c}$$

将不等式乘以 $n/2^p$，因为 $n<0$，所以它变为

$$\frac{n}{d}\frac{n_c+1}{n_c}<\frac{mn}{2^p}\leqslant\frac{n}{d}\frac{2^p+1}{2^p}$$

或者说

$$\left\lfloor\frac{n}{d}\frac{n_c+1}{n_c}\right\rfloor+1\leqslant\left\lfloor\frac{mn}{2^p}\right\rfloor+1\leqslant\left\lfloor\frac{n}{d}\frac{2^p+1}{2^p}\right\rfloor+1$$

根据定理 D2 可得：

$$\left\lceil\frac{n(n_c+1)-dn_c+1}{dn_c}\right\rceil+1\leqslant\left\lfloor\frac{mn}{2^p}\right\rfloor+1\leqslant\left\lceil\frac{n(2^p+1)-2^pd+1}{2^pd}\right\rceil+1$$

$$\left\lceil\frac{n(n_c+1)+1}{dn_c}\right\rceil\leqslant\left\lfloor\frac{mn}{2^p}\right\rfloor+1\leqslant\left\lceil\frac{n(2^p+1)+1}{2^pd}\right\rceil$$

由于 $n+1\leqslant 0$，所以可放宽不等式右侧：

$$\left\lceil\frac{n}{d}+\frac{n+1}{dn_c}\right\rceil\leqslant\left\lfloor\frac{mn}{2^p}\right\rfloor+1\leqslant\left\lceil\frac{n}{d}\right\rceil \tag{11}$$

当 $-n_c\leqslant n\leqslant -1$ 时，有

$$\frac{-n_c+1}{dn_c}\leqslant\frac{n+1}{dn_c}\leqslant 0,\text{或者说，}$$

$$-\frac{1}{d}<\frac{n+1}{dn_c}\leqslant 0$$

因此，根据定理 D4 可知，

$$\left\lceil\frac{n}{d}+\frac{n+1}{dn_c}\right\rceil=\left\lceil\frac{n}{d}\right\rceil$$

这样的话，等式（1b）在此情况（$-n_c\leqslant n\leqslant -1$）下就成立了。

当 $n<-n_c$ 时，n 的取值范围是

$$-n_c-d\leqslant n\leqslant -n_c-1 \tag{12}$$

（由（3a）可知，假如 $n<-n_c-d$，那么 $n<-2^{W-1}$，而这是不可能的。）用初等代数知识变换不等式（11）的左侧，可得出

$$\left\lceil\frac{-n_c-1}{d}+\frac{(n+n_c)(n_c+1)+1}{dn_c}\right\rceil\leqslant\left\lfloor\frac{mn}{2^p}\right\rfloor+1\leqslant\left\lceil\frac{n}{d}\right\rceil \tag{13}$$

在 $-n_c-d+1\leqslant n\leqslant -n_c-1$ 的情况下，

$$\frac{(-d+1)(n_c+1)}{dn_c}+\frac{1}{dn_c}\leqslant\frac{(n+n_c)(n_c+1)+1}{dn_c}\leqslant\frac{-(n_c+1)+1}{dn_c}=-\frac{1}{d}$$

当 n_c 最小时，比值 $(n_c+1)/n_c$ 最大；也就是说，在 $n_c=d-1$ 时比值最大。因此：

$$\frac{(-d+1)(d-1+1)}{d(d-1)}+\frac{1}{dn_c}\leqslant\frac{(n+n_c)(n_c+1)+1}{dn_c}<0$$

$$-1<\frac{(n+n_c)(n_c+1)+1}{dn_c}<0$$

因为 $(-n_c-1)/d$ 是整数，而与它相加的量又在 0 与 -1 之间，所以，从（13）式可知：

$$\frac{-n_c-1}{d}\leqslant\left\lfloor\frac{mn}{2^p}\right\rfloor+1\leqslant\left\lceil\frac{n}{d}\right\rceil$$

对于 $-n_c-d+1\leqslant n\leqslant -n_c-1$ 范围内的 n 来说，

$$\left\lceil\frac{n}{d}\right\rceil=\frac{-n_c-1}{d}$$

因此可得 $\lfloor mn/2^p\rfloor+1=\lceil n/d\rceil$，于是（1b）成立。

最后一种情况 $n=-n_c-d$ 只有在 d 为某些特定值时才可能发生。从（3a）可知 $-n_c-d\leqslant -2^{W-1}$，所以，如果 n 取这个值，那么，$n=-n_c-d=-2^{W-1}$，因此可得 $n_c=2^{W-1}-d$。这样的话，$\text{rem}(2^{W-1}, d)=\text{rem}(n_c+d, d)=d-1$（也就是说，$2^{W-1}+1$ 能被 d 整除）。

在这种情况（$n=-n_c-d$）下，$p=W-1$（p 可能取到的最小值）是（6）式的解，原因在于，当 $p=W-1$ 时，

$$\begin{aligned}n_c(d-\text{rem}(2^p,d))&=(2^{W-1}-d)(d-\text{rem}(2^{W-1},d))\\&=(2^{W-1}-d)(d-(d-1))=2^{W-1}-d<2^{W-1}=2^p\end{aligned}$$

从（5）式可知：

$$m=\frac{2^{W-1}+d-\text{rem}(2^{W-1},d)}{d}=\frac{2^{W-1}+d-(d-1)}{d}=\frac{2^{W-1}+1}{d}$$

因此，

$$\begin{aligned}\left\lfloor\frac{mn}{2^p}\right\rfloor+1&=\left\lfloor\frac{2^{W-1}+1}{d}\frac{-2^{W-1}}{2^{W-1}}\right\rfloor+1=\left\lfloor\frac{-2^{W-1}-1}{d}\right\rfloor+1\\&=\left\lceil\frac{-2^{W-1}-d}{d}\right\rceil+1=\left\lceil\frac{-2^{W-1}}{d}\right\rceil=\left\lceil\frac{n}{d}\right\rceil\end{aligned}$$

所以（1b）成立。

综上可证：如果分别按照（5）式和（6）式计算出 m 与 p，那么等式（1a）与（1b）对所有 n 都成立。

10.5 除数小于等于 -2 的带符号除法

因为带符号整数除法满足 $n\div(-d)=-(n\div d)$，所以可按照 $n\div|d|$ 来生成代码，然后用一条指令把商设为其相反数。（当 $d=-2^{W-1}$ 时不能这么做，不过由于此数的相反数是 2 的幂，所以可按照 10.1 节的方法计算，最后加一条取相反数的指令即可。）不能对被除数取相反数，因为它有可能是绝对值最大的负数。

按照下列方式计算，可以省掉取相反数指令：

$$\text{如果 } n \leqslant 0,\ q = \left\lfloor \frac{mn}{2^p} \right\rfloor$$

$$\text{如果 } n > 0,\ q = \left\lfloor \frac{mn}{2^p} \right\rfloor + 1$$

在 $n>0$ 时加 1 比较难办（因为不能直接使用 n 的符号位），不如把代码写成在 $q<0$ 时加 1。因为乘数 m 为负，所以这两种运算是等效的（如下例所示）。

当 $W=32$ 且 $d=-7$ 时，生成的代码如下：

```
li    M,0x6DB6DB6D  Magic num, -(2**34+5)/7 + 2**32.
mulhs q,M,n         q = floor(M*n/2**32).
sub   q,q,n         q = floor(M*n/2**32) - n.
shrsi q,q,2         q = floor(q/4).
shri  t,q,31        Add 1 to q if
add   q,q,t         q is negative (n is positive).

muli  t,q,-7        Compute remainder from
sub   r,n,t         r = n - q*(-7).
```

此代码与除以 +7 时所用的一样，区别在于：此乘数与除以 +7 时所用的乘数互为相反数，乘法后面跟的是 sub 指令，而不是 add 指令，而且如上所述，移位量为 31 的 shri 指令移动 q，不移动 n。（在 $d=+7$ 的情况下也可以移动 q，不过会降低代码的并发执行能力。）由于操作数正负号相同，所以减法不会溢出。然而这个办法并不总能奏效。虽说上述代码在 $W=32$，$d=-7$ 时正确，但是，由“除以 3”修改而来的类似代码却无法在 $W=32$，$n=-2^{31}$ 的情况下正确算出 n 除以 -3 的值。

现在详细看看此情况。

给定字长 W 和除数 d，满足 $W \geqslant 3$ 且 $-2^{W-1} \leqslant d \leqslant -2$，要寻找绝对值最小的整数 m 与整数 p，使得：

$$\text{当 } -2^{W-1} \leqslant n \leqslant 0 \text{ 时，} \left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor, \tag{14a}$$

$$\text{当 } 1 \leqslant n < 2^{W-1} \text{ 时，} \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lfloor \frac{n}{d} \right\rfloor, \tag{14b}$$

其中 $-2^W \leqslant m \leqslant 0$ 且 $p \geqslant W$。

与除数为正时类似，在 n 的取值范围内寻找绝对值最大的负数 n_c，使得 $n_c = kd+1$，其中 k 为某整数。肯定能找到这样的 n_c，因为 $n_c = d+1$ 就符合条件。可通过 $n_c = \lfloor(-2^{W-1}-1)/d\rfloor d + 1 = -2^{W-1} + \mathrm{rem}(2^{W-1}+1,\ d)$ 算出 n_c。在 n 的取值范围内最小的那 $|d|$ 个数里，必有一个是 n_c，所以

$$-2^{W-1} \leqslant n_c \leqslant -2^{W-1} - d - 1 \tag{15a}$$

而且显然可知

$$n_c \leqslant d+1 \tag{15b}$$

因为等式（14b）一定对 $n=-d$ 成立，而等式（14a）一定对 $n=n_c$ 成立，所以可得出下面这个与（4）式相仿的不等式：

$$\frac{2^p}{d}\frac{n_c - 1}{n_c} < m < \frac{2^p}{d} \tag{16}$$

由于 m 是满足（16）式的最大整数，所以它就是小于 $2^p/d$ 的下一个整数，也就是说

$$\boxed{m=\frac{2^p-d-\text{rem}(2^p,d)}{d}} \tag{17}$$

将此式与不等式（16）的左半边结合并化简，可得

$$\boxed{2^p>n_c(d+\text{rem}(2^p,d))} \tag{18}$$

可按与除数为正时类似的方法，证明由（17）式与（18）式所确定的算法可行，并证明乘积正确，此处不赘述，其中的难点在于如何证明 $-2^W\leqslant m\leqslant 0$。按照 d 的相反数是否为 2 的幂，可分两种情况证明此不等式。若 $d=-2^k$，则很容易就能证明 $n_c=-2^{W-1}+1$，$p=W+k-1$ 且 $m=-2^{W-1}-1$（这样的 m 在规定范围内）。如果 d 不是 -2^k，那么只需修改早前的证明过程，即可证明此种情况。

除数为何值时 $m(-d)\neq -m(d)$

$m(d)$ 表示除数 d 所对应的乘数。如果 $m(-d)=-m(d)$，那么除数为负的代码就可以这样来生成：先根据 $|d|$ 算出乘数，然后取其相反数，再用同前面"除以－7"时相似的办法生成除以 $-d$ 的算法代码。

将（18）式与（6）式对比，将（17）式与（5）式对比，可以看出，若除数为 $-d$ 时的 n_c 与除数为 d 时的 n_c 互为相反数，则 $m(-d)=-m(d)$。因此，只有根据负除数所计算出来的 n_c 值是绝对值最大的负数 -2^{W-1} 时，才会出现 $m(-d)\neq -m(d)$ 的情况。这样的除数，其相反数是 $2^{W-1}+1$ 的因子。由下列因数分解（数据得自 Scratchpad 软件[⊖]）可知，这种数相当罕见。

$$2^{15}+1=3^2\cdot 11\cdot 331$$
$$2^{31}+1=3\cdot 715\ 827\ 883$$
$$2^{63}+1=3^3\cdot 19\cdot 43\cdot 5419\cdot 77\ 158\ 673\ 929$$

对于所有此类因数来说，$m(-d)\neq -m(d)$。大略证明如下：当 $d>0$ 时，可知 $n_c=2^{W-1}-d$。由于 $\text{rem}(2^{W-1},d)=d-1$，所以当 $p=W-1$ 时（6）式成立，因此当 $p=W$ 时（6）也成立。然而当 $d<0$ 时，可知 $n_c=-2^{W-1}$ 且 $\text{rem}(2^{W-1},d)=|d|-1$。因此，$p=W-1$ 或 $p=W$ 都不能令（18）式成立，所以 $p>W$。

10.6 将除法算法集成至编译器中

如果想让编译器将除数为常量的除法转换为乘法，那么必须根据给定的 d 计算出"神奇数字" M 与移位量 s。直观的计算方法是根据（6）式或（18）式，依照 $p=W$，

⊖ 此软件现名 Axiom，是一款免费而通用的计算机代数系统。详情参见：http://www.axiom-developer.org/。——译者注

$W+1$，... 的顺序一直尝试下去，直至找到令该式成立的 p 值。然后，按照（5）式或（17）式计算 m。将 m 重新表示为带符号整数，即可得出 M，而 s 等于 $p-W$。

下面演示的算法可以处理 d 为正数及负数的情况，它只多增了一点点附加代码，而且没有使用双字算术。

回忆一下计算 n_c 所用的式子：

$$n_c=\begin{cases}2^{W-1}-\mathrm{rem}(2^{W-1},d)-1,\text{若 } d>0\\-2^{W-1}+\mathrm{rem}(2^{W-1}+1,d),\text{若 } d<0\end{cases}$$

由此看来，$|n_c|$ 可按如下方式算出：

$$t=2^{W-1}+\begin{cases}0,\text{若 } d>0\\1,\text{ 若 } d<0\end{cases}$$

$$|n_c|=t-1-\mathrm{rem}(t,|d|)$$

鉴于求余运算的参数取值，这里必须用无符号除法来算余数。上式中没有写 $\mathrm{rem}(t,d)$，而是写成了 $\mathrm{rem}(t,|d|)$，就是要强调程序必须处理两个值为正的无符号参数。

由（6）式及（18）式可知，p 值可按下式算出：

$$2^p>|n_c|(|d|-\mathrm{rem}(2^p,|d|)) \tag{19}$$

其后可根据（5）式及（17）式算出 $|m|$：

$$|m|=\frac{2^p+|d|-\mathrm{rem}(2^p,|d|)}{|d|} \tag{20}$$

若想直接求出（19）式，则需使用“长除法”（将含有 $2W$ 个二进制位的被除数除以 W 位的余数，得出 W 位的商与余数），而且实际上必须是无符号长除法才行。有种方法能求解（19）式，它不使用长除法就能执行全部运算，并且在常见的高级程序语言中仅凭 W 位算术操作即能轻松实现。不过，这种方法还是要执行无符号除法及无符号数比较操作。

我们要逐步算出 $\mathrm{rem}(2^p,|d|)$：初始化 q 与 r 两个变量，分别表示 2^p 除以 $|d|$ 的商和余数，设定 $p=W-1$，然后逐渐递增 p，并更新 q 与 r。

在每轮搜索中，p 递增 1，q 与 r 则随着搜索过程按如下方式更新（参见定理 D5 左侧算式）

```
q = 2*q;
r = 2*r;
if (r >= abs(d)) {
   q = q + 1;
   r = r - abs(d);}
```

根据不等式（4）的左半边与不等式（16）的右半边，再结合已经证明过的 m 取值边界，可以得出 $q=\lfloor 2^p/|d|\rfloor<2^W$，因此 q 可以用 W 位无符号整数来表示。又因为 $0\leqslant r<|d|$，所以 r 可用 W 位带符号或无符号整数来表示。（由于中间结果 $2r$ 可能超过 $2^{W-1}-1$，所以 r 应该是无符号数，而上述代码中的比较操作也应是无符号的。）

接下来计算 $\delta=|d|-r$。减法中的两项都可以用 W 位无符号整数表示，其结果也如

此（因为 $1 \leqslant \delta \leqslant |d|$），所以这里不会出问题。

为了避免（19）式中的长除法，可以将其变形为：

$$\frac{2^p}{|n_c|} > \delta$$

$2^p/|n_c|$ 这个量可表示成 W 位无符号整数（与（7）式类似，由（19）式可证明 $2^p \leqslant 2|n_c| \cdot |d|$，而且因为在 $d=-2^{W-1}$ 时，$n_c=-2^{W-1}+1$ 且 $p=2^{W-2}$，所以当 $W \geqslant 3$ 时，$2^p/|n_c|=2^{2W-2}/(2^{W-1}-1)<2^W$）。使用与计算 $\text{rem}(2^p, |d|)$ 相同的方式，很容易就能在 p 的增长过程中逐步算出它的值。比较操作也应是无符号的，因为会出现 $2^p/|n_c| \geqslant 2^{W-1}$ 的情况（如果 d 较大，就有可能发生此状况）。

不要直接通过（20）式计算 m（那样做需要长除法）。观察到

$$\frac{2^p+|d|-\text{rem}(2^p, |d|)}{|d|} = \left\lfloor \frac{2^p}{|d|} \right\rfloor + 1 = q+1$$

循环结束条件"$2^p/|n_c|>\delta$"很难求值。只有通过商 q_1 和余数 r_1 才能表示出 $2^p/|n_c|$。$2^p/|n_c|$ 也许是整数，也许不是（只有当或 $d=2^{W-2}+1$ 为其他一些负值时，它才是整数）。通过如下代码可判断出 $2^p/|n_c| \leqslant \delta$：

$$\boldsymbol{q_1 < \delta \mid (q_1 = \delta \ \& \ r_1 = 0)}$$

图 10.1 列出了根据 d 计算 M 与 s 的完整 C 语言代码，它适用于 $W=32$ 的情况。有些地方会溢出，不过若是忽略溢出，则能得出正确结果。

如果想利用此程序的运算结果，那么编译器就应该生成 li 与 mulhs 指令，并在 d>0 且 M<0 时生成 add 指令，在 d<0 且 M>0 时生成 sub 指令，在 s>0 时生成 shrsi 指令。其后要生成 shri 指令，最后还需生成 add 指令。

对于 $W=32$ 的情况来说，可以不处理负除数：预先计算好 $d=3$ 和 $d=715\ 827\ 883$ 时的结果，并且用 $m(-d)=-m(d)$ 处理其他负除数。然而，这样修改即便能缩短程序，也不会比图 10.1 短太多。

```
struct ms {int M;          // Magic number
           int s;};        // and shift amount.

struct ms magic(int d) {   // Must have 2 <= d <= 2**31-1
                           // or   -2**31 <= d <= -2.
   int p;
   unsigned ad, anc, delta, q1, r1, q2, r2, t;
   const unsigned two31 = 0x80000000;     // 2**31.
   struct ms mag;

   ad = abs(d);
   t = two31 + ((unsigned)d >> 31);
   anc = t - 1 - t%ad;       // Absolute value of nc.
   p = 31;                   // Init. p.
   q1 = two31/anc;           // Init. q1 = 2**p/|nc|.
   r1 = two31 - q1*anc;      // Init. r1 = rem(2**p, |nc|).
   q2 = two31/ad;            // Init. q2 = 2**p/|d|.
   r2 = two31 - q2*ad;       // Init. r2 = rem(2**p, |d|).
   do {
```

图 10.1 计算带符号除法所用的神奇数字

```
      p = p + 1;
      q1 = 2*q1;            // Update q1 = 2**p/|nc|.
      r1 = 2*r1;            // Update r1 = rem(2**p, |nc|).
      if (r1 >= anc) {      // (Must be an unsigned
         q1 = q1 + 1;       // comparison here.)
         r1 = r1 - anc;}
      q2 = 2*q2;            // Update q2 = 2**p/|d|.
      r2 = 2*r2;            // Update r2 = rem(2**p, |d|).
      if (r2 >= ad) {       // (Must be an unsigned
         q2 = q2 + 1;       // comparison here.)
         r2 = r2 - ad;}
      delta = ad - r2;
   } while (q1 < delta || (q1 == delta && r1 == 0));

   mag.M = q2 + 1;
   if (d < 0) mag.M = -mag.M; // Magic number and
   mag.s = p - 32;            // shift amount to return.
   return mag;
}
```

图 10.1 （续）

10.7 其他主题

定理 DC2 若不强令 p 等于 W，则最小乘数 m 是奇数。

证明：假定 p 是满足等式（1a）与（1b）的最小整数（不强令其等于 W），而 m 为偶数。由此显然可见：将 m 除以 2 并且将 p 减 1 之后，新的 m 与 p 仍能满足等式（1a）与（1b）。这与 p 为满足两等式的最小整数这一假设相矛盾。■

10.7.1 唯一性

与给定除数相对应的神奇数字有时是唯一的（例如当 $W=32$ 且 $d=7$ 时），但常常并非如此。实际上，实验表明，神奇数字通常并不唯一。例如，当 $W=32$ 且 $d=6$ 时，有 4 个神奇数字：

$$
\begin{aligned}
M &= 715\ 827\ 883((2^{32}+2)/6), & s&=0\\
M &= 1\ 431\ 655\ 766((2^{32}+2)/3), & s&=1\\
M &= -1\ 431\ 655\ 765((2^{33}+1)/3-2^{32}), & s&=2\\
M &= -1\ 431\ 655\ 764((2^{33}+4)/3-2^{32}), & s&=2
\end{aligned}
$$

不过，神奇数字有如下唯一性规律：

定理 DC3 对于给定的除数 d 来说，若不强令 p 等于 W，则只有一个乘数 m 可使 p 值最小。

证明：首先考虑 $d>0$ 的情况。不等式（4）的上下界之差为 $2^p/dn_c$。我们已经证明了（7）式，也就是说，如果 p 最小，那么 $2^p/dn_c \leqslant 2$。因此，至多有两个 m 符合（4）式。设 m 是由（5）式所确定的两个取值中的小者，$m+1$ 是大者。

设 p_0 是令 $m+1$ 满足（4）式右半边的所有 p 值里最小的那个（不强令 p_0 等于 W）。则有

$$\frac{2^{p_0}+d-\mathrm{rem}(2^{p_0},d)}{d}+1<\frac{2^{p_0}}{d}\frac{n_c+1}{n_c}$$

此式可化简为

$$2^{p_0}>n_c(2d-\mathrm{rem}(2^{p_0},d))$$

将不等式两边除以 2，可得

$$2^{p_0-1}>n_c\left(d-\frac{1}{2}\mathrm{rem}(2^{p_0},d)\right)$$

因为 $\mathrm{rem}(2^{p_0},d)\leqslant 2\mathrm{rem}(2^{p_0-1},d)$（根据 9.1 节定理 D5），所以

$$2^{p_0-1}>n_c(d-\mathrm{rem}(2^{p_0-1},d))$$

这与 p_0 在所有 p 值中最小这一假设相矛盾。

证明 $d<0$ 时的步骤与此类似，不再列出。 ■

10.7.2 可生成最佳程序代码的除数

适用于 $d=3$，$W=32$ 的除法程序很短，因为 mulhs 指令后面不再需要 add 或 shrsi 指令了。除数为其他值时也能生成如此短小的程序吗？

只考虑除数为正的情况。我们要寻找满足等式（1a）与（1b）的整数 m 和 p，且两数要满足 $p=W$ 与 $0\leqslant m<2^{W-1}$。由于满足等式（1a）与（1b）的整数 m 和 p 必然也满足（4）式，所以只要在 $p=W$ 和 $0\leqslant m<2^{W-1}$ 这两个条件下求解（4）式，即可找出这些除数了。在 $p=W$ 时，（4）式的解如下：

$$m=\frac{2^W+kd-\mathrm{rem}(2^W,d)}{d},k=1,2,3,\ldots$$

将此式与不等式（4）右半边结合并化简，可得

$$\mathrm{rem}(2^W,d)>kd-\frac{2^W}{n_c} \tag{21}$$

当 $k=1$ 且 n_c 取其最小值 2^{W-2} 时，$\mathrm{rem}(2^W,d)$ 的取值范围最宽松。因此，下式必然成立：

$$\mathrm{rem}(2^W,d)>d-4$$

也就是说，2^W+1、2^W+2 或 2^W+3 能够被 d 整除。

现在看看在这 3 个数中，当除数 d 是哪个数的因子时，所生成的代码最优。

如果 2^W+1 能被 d 整除，则 $\mathrm{rem}(2^W,d)=d-1$。这样的话，$p=W$ 就是（6）式的一个解了，因为不等式变成了

$$2^W>n_c(d-(d-1))=n_c$$

由于 $n_c<2^{W-1}$，所以上式显然成立。然后计算 m：

$$m=\frac{2^W+d-(d-1)}{d}=\frac{2^W+1}{d}$$

当 $d\geqslant 3$ 时（由于 d 必须能整除 2^W+1，所以 $d\neq 2$），m 小于 2^{W-1}。因此，只要除数 d 是 2^W+1 的因子，就能生成最优代码。

与之类似，如果 d 能整除 2^W+2，那么 $\mathrm{rem}(2^W,d)=d-2$。此时 $p=W$ 依然为（6）

式的一个解，因为不等式变成了

$$2^W > n_c(d-(d-2)) = 2n_c$$

此式显然成立。然后计算 m：

$$m = \frac{2^W + d - (d-2)}{d} = \frac{2^W + 2}{d}$$

当时 $d=2$，这个值超过了 $2^{W-1}-1$，但是当 $W \geqslant 3$ 且 $d \geqslant 3$ 时，该值小于等于 $2^{W-1}-1$（不会发生 $W=3$ 且 $d=3$ 的情况，因为 3 不是 $2^3+2=10$ 的因子）。因此，如果除数 d 是 2^W+2 的因子，同时 d 本身不为 2，也不是 2 的"余因子"（cofactor），那么就能生成最优代码来执行这样的除法。（对于 2^W+2 来说，2 的"余因子"是 $(2^W+2)/2$，这个数不能表示为 W 位带符号整数）。

如果 d 能整除 2^W+3，那么接下来的论证过程将表明此时无法生成最优代码。由于 $\text{rem}(2^W, d) = d-3$，所以根据不等式（21）可知，对 $k=1, 2, 3, \ldots$ 来说，下列不等式成立：

$$n_c < \frac{2^W}{kd-d+3}$$

在 $k=1$ 时 n_c 的取值范围最宽松，所以必有 $n_c < 2^W/3$。

由不等式（3a）式可知，$n_c \geqslant 2^{W-1}-d$ 或者说 $d \geqslant 2^{W-1}-n_c$。因此，下式必然成立：

$$d > 2^{W-1} - \frac{2^W}{3} = \frac{2^W}{6}$$

由于 2、3、4 都无法整除 2^W+3，所以 2^W+3 可能有的最小因子是 5。因此，可能有的最大因子就是 $(2^W+3)/5$。因此，如果 d 能整除 2^W+3，而且可以生成除以 d 所用的最优代码，那么必有

$$\frac{2^W}{6} < d \leqslant \frac{2^W+3}{5}$$

对此不等式取倒数，然后乘以 2^W+3，就可看出 d 的余因子 $(2^W+3)/d$ 在如下范围内取值：

$$5 \leqslant \frac{2^W+3}{d} < \frac{(2^W+3)\cdot 6}{2^W} = 6 + \frac{18}{2^W}$$

在 $W \geqslant 5$ 时，此式意味着余因子只可能是 5 和 6。对于 $W<5$ 的情况来说，很容易就能验证 2^W+3 没有因子[㊀]。由于 6 不可能是 2^W+3 的因子，所以唯一可能的因子就是 5。因此，在 2^W+3 的因子中，有可能生成最优代码的只可能是 $(2^W+3)/5$。

当 $d=(2^W+3)/5$ 时，有

$$n_c = \left\lfloor \frac{2^{W-1}}{(2^W+3)/5} \right\rfloor \left(\frac{2^W+3}{5}\right) - 1$$

对于 $W \geqslant 4$ 的情况来说，有

㊀ 这里的"因子"（factor）指除去 1 和该数本身之外的其他正整数。——译者注

$$2<\frac{2^{W-1}}{(2^W+3)/5}<2.5$$

所以

$$n_c=2\left(\frac{2^W+3}{5}\right)-1$$

这超过了 $(2^W/3)$，与刚才的推导结果相矛盾，所以 $d=(2^W+3)/5$ 时不能生成最优代码。因为小于 4 的 W 不可能是 2^W+3 的因子，所以，综上可证：在除数 d 为 2^W+3 的因子时，无法生成最优代码。

总之，对于 2^W+1 与 2^W+2 的所有因子（除去 2 和 $(2^W+2)/2$ 之外）来说，都能生成最优代码，而当除数 d 为其他数字时，则无法生成最优代码。此外，上述论证过程也说明，在有可能生成最优代码时，"神奇数字算法"（algorithm magic，即 10.6 节的图 10.1）总能生成这样的代码。

现在看看 $W=16$，32 和 64 时的特例。相关情况的因数分解如下。

$2^{16}+1=65\,537$（质数）　　　$2^{32}+1=641\cdot 6\,700\,417$

$2^{16}+2=2\cdot 3^2\cdot 11\cdot 331$　　　$2^{32}+2=2\cdot 3\cdot 715\,827\,883$

$2^{64}+1=274\,177\cdot 67\,280\,421\,310\,721$

$2^{64}+2=2\cdot 3^3\cdot 19\cdot 43\cdot 5419\cdot 77\,158\,673\,929$

$W=16$ 时的结果表明，有 20 种除数都能生成最优代码。其中小于 100 的是：3，6，9，11，18，22，33，66，99。

在 $W=32$ 时，有 6 个除数可生成最优代码，其分别为：3，6，641，6，700，417，715，827，883，1，431，655，766。

在 $W=64$ 时，有 126 个这样的除数，其中小于 100 的是：3，6，9，18，19，27，38，43，54，57，86。

10.8 无符号除法

如果无符号除法的除数为 2 的幂，那么当然只用一条逻辑右移指令就能实现，余数可以用"同立即数求与"（and immediate）指令得出。

想要处理其他余数，看上去似乎也很容易：只要采用 $d>0$ 时的无符号除法的结果即可，省去"当余数为负时加 1"这两条指令。接下来就会看到，无符号除法的某些细节实际上较此更为复杂。

10.8.1 除数为 3 的无符号除法

当除数不是 2 的幂时，我们先来考虑在 32 位计算机上执行除以 3 的无符号除法。由于被除数最大可能为 $2^{32}-1$，所以用 $(2^{32}+2)/3$ 当乘数不合适，因为误差项 $2n/3\cdot 2^{32}$（参见 10.3.1 节中除数为 3 的例子）会超过 1/3。然而，用 $(2^{33}+1)/3$ 当乘数却可以。

代码如下：

```
li     M,0xAAAAAAAB  Load magic number, (2**33+1)/3.
mulhu q,M,n          q = floor(M*n/2**32).
shri  q,q,1

muli  t,q,3          Compute remainder from
sub   r,n,t          r = n - q*3.
```

需要用一条指令求出 64 位无符号乘积的高权重 32 位，上述代码以 mulhu 来表示。

为了说明此代码正确，我们注意到它计算的是：

$$q=\left\lfloor\frac{2^{33}+1}{3}\frac{n}{2^{33}}\right\rfloor=\left\lfloor\frac{n}{3}+\frac{n}{3\cdot 2^{33}}\right\rfloor$$

当 $0\leqslant n<2^{32}$ 时，有 $0\leqslant n/(3\cdot 2^{33})<1/3$，因此根据定理 D4 可知，$q=\lfloor n/3\rfloor$。

计算余数时，若将操作数视为带符号数，则“与立即数相乘”（multiply immediate）这一指令就有可能溢出，不过，若将操作数及结果均视为无符号数，则它就不溢出。减法指令也不可能溢出，由于其结果在 0 与 2 之间，所以余数正确。

10.8.2 除数为 7 的无符号除法

在 32 位计算机上执行除以 7 的无符号除法时，用 $(2^{32}+3)/7$，$(2^{33}+6)/7$，$(2^{34}+5)/7$ 当乘数都不合适，因为它们产生的误差项都太大了。用 $(2^{35}+3)/7$ 当乘数可以，但是它无法容纳于 32 位无符号字组中。可以先乘以 $(2^{35}+3)/7-2^{32}$，然后插入 add 指令修正乘积，这样就等于和这个大数相乘了。代码是：

```
li     M,0x24924925  Magic num, (2**35+3)/7 - 2**32.
mulhu q,M,n          q = floor(M*n/2**32).
add   q,q,n          Can overflow (sets carry).
shrxi q,q,3          Shift right with carry bit.

muli  t,q,7          Compute remainder from
sub   r,n,t          r = n - q*7.
```

此时有个问题：add 指令可能会溢出。有鉴于此，我们新发明了一条“扩展版按立即数右移指令”（shift right extended immediate，shrxi），它把 add 指令产生的进位和寄存器 q 合起来当做一个 33 位量，并将其右移，空位用 0 填充。在 Motorola 68000 系列处理器[⊖]上，可用两条指令实现此功能：以“扩展版循环移位”（*rotate with extend right*，roxr）指令右移 1 位，再逻辑右移 3 位（roxr 指令实际上使用的是“X 标志位”（X bit），然而这种 CPU 的 add 指令也会像设置进位标志那样设置“X 标志位”）。在大多数计算机上，所需指令数更多。例如，PowerPC 需要 3 条指令：先清除 q 最右侧 3 位，然后把进位标志加到 q 中，最后循环右移 3 位。

如果能设法运用 shrxi 指令，那么上述代码计算的就是：

⊖ 也称 MC68000，是由摩托罗拉公司研发的 16/32 位 CISC（复杂指令集）微处理器，于 1979 年问世，此架构目前在嵌入式领域仍有应用。详情参见：http：//zh.wikipedia.org/wiki/摩托罗拉_68000。——译者注

$$q=\left\lfloor \left(\left\lfloor \left(\frac{2^{35}+3}{7}-2^{32}\right)\frac{n}{2^{32}}\right\rfloor+n\right)/2^3 \right\rfloor=\left\lfloor \frac{n}{7}+\frac{3n}{7\cdot 2^{35}}\right\rfloor$$

当 $0\leqslant n<2^{32}$时，$0\leqslant 3n/(7\cdot 2^{35})<1/7$，所以根据定理 D4 可知 $q=\lfloor n/7\rfloor$。

Granlund 与 Montgomery [GM] 想出了一个更巧妙的方法，可以避免 shrxi 指令。这个方法用的指令数与上面所讲的以 3 条指令实现 shrxi 的方法相同，然而只用到了几乎每台计算机都有的基本指令，而且不溢出。此方法使用下列等式：

$$\left\lfloor \frac{q+n}{2^p}\right\rfloor=\left\lfloor \left(\left\lfloor \frac{n-q}{2}\right\rfloor+q\right)/2^{p-1}\right\rfloor, p\geqslant 1$$

将此式运用于本例，取 $q=\lfloor Mn/2^{32}\rfloor$，其中 $0\leqslant M<2^{32}$，这样减法就不会溢出了，因为

$$0\leqslant q=\left\lfloor \frac{Mn}{2^{32}}\right\rfloor\leqslant n$$

所以显然有 $0\leqslant n-q<2^{32}$。加法也不会溢出，因为

$$\left\lfloor \frac{n-q}{2}\right\rfloor+q=\left\lfloor \frac{n-q}{2}+q\right\rfloor=\left\lfloor \frac{n+q}{2}\right\rfloor$$

且 $0\leqslant n$，$q<2^{32}$。

用这个方法可以编写下列代码，计算除数为 7 的无符号除法：

```
li     M,0x24924925  Magic num, (2**35+3)/7 - 2**32.
mulhu q,M,n          q = floor(M*n/2**32).
sub    t,n,q         t = n - q.
shri   t,t,1         t = (n - q)/2.
add    t,t,q         t = (n - q)/2 + q = (n + q)/2.
shri   q,t,2         q = (n+Mn/2**32)/8 = floor(n/7).

muli   t,q,7         Compute remainder from
sub    r,n,t         r = n - q*7.
```

假想的 shrxi 指令的移位量必须大于 0，这样的话此算法才能奏效。可以证明：若 $d>1$ 且乘数 $m\geqslant 2^{32}$（此时就需要使用 shrxi 指令了），则移位量大于 0。

10.9 除数大于等于 1 的无符号除法

给定字长 W 与除数 d，其中 $W\geqslant 1$，$1\leqslant d<2^W$，现在需要找到符合 $0\leqslant m<2^{W+1}$ 与 $p\geqslant W$ 的最小整数 m 与整数 p，使得下式在 $0\leqslant n<2^W$时成立：

$$\left\lfloor \frac{mn}{2^p}\right\rfloor=\left\lfloor \frac{n}{d}\right\rfloor 或\ 0\leqslant n\leqslant 2^W \tag{22}$$

在做无符号除法时，神奇数字由下式来确定：

$$M=\begin{cases} m, & 若\ 0\leqslant m\leqslant 2^W, \\ m-2^W, & 若\ 2^W\leqslant m\leqslant 2^{W+1} \end{cases}$$

由于 (22) 式必然在 $n=d$ 时成立，所以有$\lfloor md/2^p\rfloor=1$，或者说

$$\frac{md}{2^p}\geqslant 1 \tag{23}$$

与带符号除法一样，在 n 的取值范围中找出一个最大的数 n_c，它必须满足 $\mathrm{rem}(n_c, d)=d-1$ 的最大数。用 $n_c=\lfloor 2^W/d \rfloor d-1=2^W-\mathrm{rem}(2^W, d)-1$ 可算出其值。于是可得

$$2^W-d \leqslant n_c \leqslant 2^W-1 \tag{24a}$$

及

$$n_c \geqslant d-1 \tag{24b}$$

由此可知 $n_c \geqslant 2^{W-1}$。

由于（22）式在 $n=n_c$时成立，所以有

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c-(d-1)}{d}$$

或者说

$$\frac{mn_c}{2^p} < \frac{n_c+1}{d}$$

将此式与式（23）结合，可得

$$\frac{2^p}{d} \leqslant m < \frac{2^p}{d}\frac{n_c+1}{n_c} \tag{25}$$

由于 m 是满足式（25）的最小整数，所以它是大于等于 $2^p/d$ 的下一个整数，也就是说：

$$\boxed{m=\frac{2^p+d-1-\mathrm{rem}(2^p-1, d)}{d}} \tag{26}$$

将此式与式（25）右半边结合并化简，可得

$$\boxed{2^p > n_c(d-1-\mathrm{rem}(2^p-1, d))} \tag{27}$$

10.9.1 无符号版算法

由上可知，算法实际上是通过试错法找到大于等于 W 且符合（27）式的最小 p 值。然后，根据（26）式算出 m。此 m 就是其可能取到的最小值，在 $p \geqslant W$ 的条件它下满足（22）式。与带符号除法时的情况类似，如果（27）式对某个 p 值成立，那么它对所有更大 p 值也成立。证明过程其实和推导定理 DC1 时相似，只不过这次会用到定理 D5（b）而非定理 D5（a）。

10.9.2 算法可行性证明

必须证明（27）式总有解，且根据其值算出来的 m 符合 $0 \leqslant m < 2^W+1$。

对任意非负整数 x 来说，必然存在某个 2 的幂，其值大于 x 且小于等于 $2x+1$，所以根据（27）式可得：

$$n_c(d-1-\mathrm{rem}(2^p-1, d)) < 2^p \leqslant 2n_c(d-1-\mathrm{rem}(2^p-1, d))+1$$

因为 $0 \leqslant \mathrm{rem}(2^p-1, d) \leqslant d-1$，所以

$$1 \leqslant 2^p \leqslant 2n_c(d-1)+1 \tag{28}$$

因为 n_c，$d \leqslant 2^W-1$，所以上式变成

$$1 \leqslant 2^p \leqslant 2(2^W-1)(2^W-2)+1$$

或者说

$$0 \leqslant p \leqslant 2W \tag{29}$$

因此，(27) 式总有解。

如果不强令 p 等于 W，那么根据 (25) 式及 (28) 式可得：

$$\frac{1}{d} \leqslant m < \frac{2n_c(d-1)+1}{d}\frac{n_c+1}{n_c}$$

$$1 \leqslant m < \frac{2d-2+1/n_c}{d}(n_c+1)$$

$$1 \leqslant m < 2(n_c+1) \leqslant 2^{W+1}$$

如果强令 p 等于 W，那么根据 (25) 式可得：

$$\frac{2^W}{d} \leqslant m < \frac{2^W}{d}\frac{n_c+1}{n_c}$$

由于 $1 \leqslant d \leqslant 2^W-1$ 且 $nc \geqslant 2^{W-1}$，所以

$$\frac{2^W}{2^W-1} \leqslant m < \frac{2^W}{1}\frac{2^{W-1}+1}{2^{W-1}}$$

$$2 \leqslant m \leqslant 2^W+1$$

不论哪种情况，m 都在“除数为 7 的无符号除法”那个范例代码的限制范围内。

10.9.3 证明无符号版算法的乘积正确

必须证明：如果 p 和 m 分别根据 (27) 式及 (26) 式算出，那么这两个值可令 (22) 式成立。

由等式 (26) 及不等式 (27) 很容易就能看出，若此二式成立，则 (25) 式必定成立。不等式 (25) 与不等式 (4) 几乎一样，证明余数正确的过程也差不多和 $n \geqslant 0$ 时证明带符号数除法所用的方法相同。

10.10 将无符号除法算法集成至编译器中

直接根据刚才证明的表达式来实现算法会比较难。尽管已经证明了 $p \leqslant 2W$，但仍有可能发生 $p=2W$ 的情况（例如，$d=2^W-2$ 而 $W \geqslant 4$ 时）。在 $p=2W$ 时，由于 (26) 式中的被除数不能放在 $2W$ 位的字组中，所以很难计算 m。

不过，可以用“神奇数字算法”(algorithm magic) 中的“逐步求商和余数”(incremental division and reminder) 这一技巧来实现。图 10.2 列出了 $W=32$ 时的算法。此算

法会回传一个名为 a 的指示器，用以表示是否需要生成 add 指令。（在执行带符号除法的情况下，调用者可通过 M 与 d 的正负号是否相反来判断这一点。）

```
struct mu {unsigned M;     // Magic number,
          int a;           // "add" indicator,
          int s;};         // and shift amount.

struct mu magicu(unsigned d) {
                           // Must have 1 <= d <= 2**32-1.
   int p;
   unsigned nc, delta, q1, r1, q2, r2;
   struct mu magu;

   magu.a = 0;             // Initialize "add" indicator.
   nc = -1 - (-d)%d;       // Unsigned arithmetic here.
   p = 31;                 // Init. p.
   q1 = 0x80000000/nc;     // Init. q1 = 2**p/nc.
   r1 = 0x80000000 - q1*nc;// Init. r1 = rem(2**p, nc).
   q2 = 0x7FFFFFFF/d;      // Init. q2 = (2**p - 1)/d.
   r2 = 0x7FFFFFFF - q2*d; // Init. r2 = rem(2**p - 1, d).
   do {
      p = p + 1;
      if (r1 >= nc - r1) {
         q1 = 2*q1 + 1;            // Update q1.
         r1 = 2*r1 - nc;}          // Update r1.
      else {
         q1 = 2*q1;
         r1 = 2*r1;}
      if (r2 + 1 >= d - r2) {
         if (q2 >= 0x7FFFFFFF) magu.a = 1;
         q2 = 2*q2 + 1;            // Update q2.
         r2 = 2*r2 + 1 - d;}       // Update r2.
      else {
         if (q2 >= 0x80000000) magu.a = 1;
         q2 = 2*q2;
         r2 = 2*r2 + 1;}
      delta = d - 1 - r2;
   } while (p < 64 &&
           (q1 < delta || (q1 == delta && r1 == 0)));

   magu.M = q2 + 1;        // Magic number
   magu.s = p - 32;        // and shift amount to return
   return magu;            // (magu.a was set above).
}
```

图 10.2 计算无符号除法所用的神奇数字

下列关键点有助于理解此算法：

- 算法中有些地方会发生无符号溢出，这些溢出应该忽略。
- $n_c=2^W-\text{rem}(2^W,d)-1=(2^W-1)-\text{rem}(2^W-d,d)$。
- 计算 2^p 除以 n_c 的商和余数时，不能像"神奇数字算法"那样更新其值，因为此算法中的 2 * r1 可能溢出。因此，算法把本来更为自然的判断语句"if (2 * r1 >= nc)"写成了"if (r1 >= nc - r1)"。在计算 2^p-1 除以 d 的余数和商时也类似。
- 因为 $0\leqslant\delta\leqslant d-1$，所以 δ 可表示为 32 位无符号整数。
- $m=(2^p+d-1-\text{rem}(2^p-1,d))/d=\lfloor(2^p-1)/d\rfloor+1=q_2+1$。
- 算法程序中没有明确体现出"当神奇数字 M 超过 2^W-1 时将其减去 2^W"这一操作，在计算 q2 发生溢出时会自动处理此问题。

- 要设置“add 指令指示器” magu.a 的值，不能直接比较 M 与 2^{32}，也不能直接比较 q2 与 $2^{32}-1$，因为那样做有溢出问题。程序在 q2 可能溢出前预先检测其值。如果 q2 将达到 $2^{32}-1$，也就是说 M 将会大于等于 2^{32}，那么就把 magu.a 设为 1。如果 q2 在 $2^{32}-1$ 以下，则保留 magu.a 的初始值 0 不变。
- 不等式（27）与 $2^p/n_c>\delta$ 等价。
- 在循环的终止条件里要测试 p<64，如果不加这一条，那么 q1 溢出后，程序就会由于循环次数太多而算出错误结果。

为了利用此程序的结果，编译器要生成 li 与 mulhu 指令，而且，如果“add 指令指示器” a 等于 0，那么还要参照 10.8.1 节“除数为 3 的无符号除法”这一例子，生成移位量为 s 的 shri 指令（假如 s>0 的话）。如果 a=1 而计算机有 shrxi 指令，那么编译器应该参照 10.8.2 节“除数为 7 的无符号除法”这一例子，生成 add 指令，并生成移位量为 s 的 shrxi 指令。如果 a=1 而计算机没有 shrxi 指令，那么应该参照 10.8.2 节所举的例子，生成 sub 指令、移位量为 1 的 shri 指令、add 指令，最后生成移位量为 s−1 的 shri 指令（如果 s−1>0；这里的 s 不会为 0 只有一种简单情况例外，那就是除数为 1，而此时编译器可以删去除以 1 这个操作）。

10.11 与无符号除法相关的其他话题

定理 DC2u 若不强令 p 等于 W，则最小乘数 m 是奇数。

定理 DC3u 给定除数 d，若不强令 p 等于 W，则只有一个乘数 m 可使 p 值最小。

这些定理的证明过程与带符号情况下的相应定理类似。

10.11.1 可生成最佳无符号除法代码的除数

对于无符号除法来说，可用与带符号除法（参见 10.7.2 节）类似的方法分析，以便找出能够生成最优代码的除数（如果有的话），也就是只需两条指令即可求商（li, mulhu）的那种除数。分析结果是：除了 $d=1$ 的情况外，只要除数是 2^W 或 2^W+1 的因子，就能生成最优代码。对于常用的字长来说，仅有少数几个不那么显而易见的[⊖]除数可以生成最优的无符号除法代码。当 $W=16$ 时，没有此种除数。当 $W=32$ 时，只有两个除数：641 和 6 700 417。当 $W=64$ 时，依然只有两个：274 177 与 67 280 421 310 721。

$d=2^k$，$k=1, 2, \ldots$ 的情况值得一提。此时，“无符号神奇数字算法”（algorithm magicu）产生的结果是 $p=W$（已经强令 p 与 W 相等），$m=2^{32-k}$。这是 m 的最小值，然而却不是 M 的最小值。如果令 p 等于 $W+k$，那么只需充分化简，便可产生更好的代码。

⊖ 原文为 nontrivial，译为“非平凡”，是个数学概念，指结构不太简单、不太明显，因而值得研究的那种数学对象，反义词为“平凡”（trivial）。这两个词与日常用语含义不同。详情参见：http://zh.wikipedia.org/wiki/平凡_（数学）。——译者注

这样的话，$m=2^W$，$M=0$，$a=1$，$s=k$。生成的代码中有乘以0的操作，于是简化成一条右移k位的指令即可。在实际使用中，如果除数是2的幂，那么似乎应视为特例，不使用magicu算法。（带符号除法不会出现这种现象，因为带符号除法所用的m不可能是2的幂。证明如下：当$d>0$时，将不等式（4）与（3b）结合起来，可得$d-1<2^p/m<d$。因此$2^p/m$不可能是整数。当$d<0$时，把不等式（16）与（15b）结合起来，亦有类似推导结果。）

对无符号除法来说，如果计算机不支持shrxi指令，那么在$m\geqslant 2^W$的情况下生成的代码要比$m<2^W$时糟糕得多。我们不妨来看看需要大乘数的频率有多高。当$W=32$时，在小于等于100的整数里，有31个这种“坏”除数（“bad” divisor）：1、7、14、19、21、27、28、31、35、37、38、39、42、45、53、54、55、56、57、62、63、70、73、74、76、78、84、90、91、95、97。

10.11.2 带符号乘法与无符号乘法互化

如果计算机没有mulhu指令，但却有mulhs指令（或者带符号长乘法指令），那么8.3节所介绍的“带符号与无符号高权重积互化”技巧也许对此处的无符号常量除数除法有所帮助。

那一节介绍了一种需要7条指令的方法，能够以mulhs来实现mulhu。不过对于此处的用法来说，由于神奇数字M已知，所以该方法可以化简。这样一来，编译器就可以根据神奇数字的最高有效位来生成如下代码，并以之取代“mulhu q, M, n”操作了，其中t表示临时寄存器：

```
  M31 = 0              M31 = 1
mulhs q,M,n          mulhs q,M,n
shrsi t,n,31         shrsi t,n,31
and   t,t,M          and   t,t,M
add   q,q,t          add   t,t,n
                     add   q,q,t
```

将与mulhu一起使用的其他指令算上，在不支持无符号乘法的计算机中，此方法需要6至8条指令来求得除数为常量的无符号除法之商。

这项技巧可以反过来用，以mulhu实现出mulhs。代码同上，只需把mulhs指令换成mulhu，并且将每一列中的add改为sub即可。

10.11.3 更简单的无符号除法生成算法

如果把神奇数字必须最小这一条去掉，那么可以得到简单些的算法。使用下式取代（27）式：

$$2^p \geqslant 2^W(d-1-\mathrm{rem}(2^p-1,d)) \tag{30}$$

然后和早前一样，用（26）式计算m。

此算法显然在形式上正确（也就是说，这样计算出来的m值确实满足等式（22）），

因为它与前述算法唯一区别在于，这样计算出来的 p 值对某些 d 来说过大了一些。可以证明，由（30）式及（26）式算出来的 m 小于 2^{W+1}。此处略去证明过程，只给出算法（参见图 10.3）。

```
struct mu {unsigned M;      // Magic number,
           int a;           // "add" indicator,
           int s;};         // and shift amount.

struct mu magicu2(unsigned d) {
                            // Must have 1 <= d <= 2**32-1.
   int p;
   unsigned p32, q, r, delta;
   struct mu magu;
   magu.a = 0;              // Initialize "add" indicator.
   p = 31;                  // Initialize p.
   q = 0x7FFFFFFF/d;        // Initialize q = (2**p - 1)/d.
   r = 0x7FFFFFFF - q*d;    // Init. r = rem(2**p - 1, d).
   do {
      p = p + 1;
      if (p == 32) p32 = 1;      // Set p32 = 2**(p-32).
      else p32 = 2*p32;
      if (r + 1 >= d - r) {
         if (q >= 0x7FFFFFFF) magu.a = 1;
         q = 2*q + 1;            // Update q.
         r = 2*r + 1 - d;        // Update r.
      }
      else {
         if (q >= 0x80000000) magu.a = 1;
         q = 2*q;
         r = 2*r + 1;
      }
      delta = d - 1 - r;
   } while (p < 64 && p32 < delta);
   magu.M = q + 1;          // Magic number and
   magu.s = p - 32;         // shift amount to return
   return magu;             // (magu.a was set above).
}
```

图 10.3 用简化过的算法计算无符号除法所用的神奇数字

下一节将谈到，Alverson 先生［Alv］给出了一个更加简单的算法，不过它算出来的 m 有些大。magicu2 算法的好处在于：当 $d \leqslant 2^{W-1}$ 时，几乎总能给出最小的 m 值。在 $W=32$ 的情况下，没能令 magicu2 算法给出最小乘数的最小除数 d 是 102，807，此时 magicu 算法算出来的 m 是 2 737 896 999 而 magic2u 算出来的 m 则是 5 475 793 997。

还有一个同 magicu2 相仿的算法适用于除数为正的带符号除法，不过对于除数为任意值的带符号除法来说，那个算法效果不是很好。

10.12 余数非负式除法与向下取整式除法的适用性

既然“若被除数为负则加 1”这步可以省去，那么想把除数为常数的“余数非负式除法”或“向下取整式除法”变为乘法看上去似乎就简单一些。实际情况并非如此。前面介绍的方法不能简单套用在“余数非负式除法”与“向下取整式除法”之上。也许能想

想办法：比如可以根据被除数的正负号，略微改动一下乘数 m。

10.13 类似算法

除了编写 magic 算法外，还可以把一些小除数所对应的神奇数字和移位量放在表里。如果除数等于表中的某数再乘以 2 的幂，那么只需按下列步骤处理即可：

1. 计算 d 的后缀 0 个数，将之记为 k。
2. 以 $d/2^k$（d 右移 k 位）为参数查表。
3. 用表中找到的数做神奇数字。
4. 将表中查得的移位量加 k 作为此次除法的移位量。

因此，如果表中含有除数 3、5、25 等，那么就可算出除数为 6、10、100 等值的除法了。

这种做法通常能给出最小神奇数字，但并不总是如此。对 $W=32$ 来说，没能令该方法给出最小神奇数字的最小正余数 d 是 334 972，此时计算出来的 m 是 3 361 176 179，而 s 是 18。实际上与 $d=334\ 972$ 对应的最小神奇数字 m 是 840 294 045，而 s 是 16。在 $d=-6$ 时，这个方法也没能给出最小神奇数字。在这两种情况下，所输出代码的质量会受影响。

不论除数为何值，接下来要讲的这个方法都能完全准确地算出除法结果，而据笔者所知，此方法由 Alverson [Alv] 首先论述。使用本书的标记法，这个无符号整除算法根据除数 d，将移位量 p 设置为 $W+\lceil \log_2 d \rceil$，将乘数 m 设置为 $\lceil 2^p/d \rceil$，然后算出除法结果 $n \div d=\lfloor mn/2^p \rfloor$（也就是用乘法和右移位指令来算）。Alverson 证明了乘数 m 小于 2^{W+1}，并证明了对于可以用 W 个二进制位来表示的所有 n 值来说，此方法都能准确算出除法的商。

Alverson 的算法是本书早前所讲算法的简化版，它无需用试错法决定 p 值，因此更适合置于硬件之中，这也是此算法作者的主要意图。他所用的乘数 m 总是大于等于 2^W，因此若以软件方式实现，则需要按照“除数为 7 的无符号除法”这一例子中的办法来编写代码（也就是说，总需要使用 add 及 shrxi 指令，或另用 4 条指令来实现这两条指令的功能）。由于大部分小除数都可以用小于 2^W 的乘数来处理，所以似乎值得寻找这些情况下的乘数。

对于带符号除法来说，Alverson 建议根据 $|d|$ 来寻找乘数 m，并且用字长为 $W-1$ 的数来表示它（也就是说 $2^{W-1} \leqslant m < 2^W$），然后将被除数乘以 m，如果除法操作的两个操作数正负号不同，则对运算结果取相反数。（乘数必须能在被除数是 2^{W-1} 时也能给出正确结果才行，2^{W-1} 是负极值的绝对值。）在乘数 m 大于等于 2^W 的情况下，按照此建议，似乎能生成比本书早前算法更好的代码。将此做法运用于除数为 7 的带符号除法，可得下述代码，其中根据关系式 $-x=\bar{x}+1$ 来避免分支指令：

```
abs    an,n
li     M,0x92492493  Magic number, (2**34+5)/7.
mulhu q,M,an         q = floor(M*an/2**32).
shri  q,q,2
shrsi t,n,31         These three instructions
xor   q,q,t          negate q if n is
sub   q,q,t          negative.
```

与早前那种“除数为 7 的带符号除法”代码相比，这样写出来的代码并不好（原来需要 6 条指令，现在需要 7 条），然而要是计算机不支持 *mulhs* 指令而支持 *abs* 与 *mulhu* 指令，那么此办法还算有用。

下节将给出一些有代表性的神奇数字。

10.14 神奇数字示例

表 10.1 和表 10.2 分别给出了 $W=32$ 和 $W=64$ 时的某些神奇数字。

表 10.1 $W=32$ 时的某些神奇数字

d	带符号除法		无符号除法		
	M（十六进制）	s	M（十六进制）	a	s
-5	99999999	1			
-3	55555555	1			
-2^k	7FFFFFFF	$k-1$			
1	—	—	0	1	0
2^k	80000001	$k-1$	2^{32-k}	0	0
3	55555556	0	AAAAAAAB	0	1
5	66666667	1	CCCCCCCD	0	2
6	2AAAAAAB	0	AAAAAAAB	0	2
7	92492493	2	24924925	1	3
9	38E38E39	1	38E38E39	0	1
10	66666667	2	CCCCCCCD	0	3
11	2E8BA2E9	1	BA2E8BA3	0	3
12	2AAAAAAB	1	AAAAAAAB	0	3
25	51EB851F	3	51EB851F	0	3
125	10624DD3	3	10624DD3	0	3
625	68DB8BAD	8	D1B71759	0	9

表 10.2 $W=64$ 时的某些神奇数字

d	带符号除法		无符号除法		
	M（十六进制）	s	M（十六进制）	a	s
-5	9999999999999999	1			
-3	5555555555555555	1			
-2^k	7FFFFFFFFFFFFFFF	$k-1$			
1	—	—	0	1	0
2^k	8000000000000001	$k-1$	2^{64-k}	0	0
3	5555555555555556	0	AAAAAAAAAAAAAAAB	0	1

（续）

d	带符号除法		无符号除法		
	M（十六进制）	s	M（十六进制）	a	s
5	6666666666666667	1	CCCCCCCCCCCCCCCD	0	2
6	2AAAAAAAAAAAAAAB	0	AAAAAAAAAAAAAAAB	0	2
7	4924924924924925	1	2492492492492493	1	3
9	1C71C71C71C71C72	0	E38E38E38E38E38F	0	3
10	6666666666666667	2	CCCCCCCCCCCCCCCD	0	3
11	2E8BA2E8BA2E8BA3	1	2E8BA2E8BA2E8BA3	0	1
12	2AAAAAAAAAAAAAAB	1	AAAAAAAAAAAAAAAB	0	3
25	A3D70A3D70A3D70B	4	47AE147AE147AE15	1	5
125	20C49BA5E353F7CF	4	0624DD2F1A9FBE77	1	7
625	346DC5D63886594B	7	346DC5D63886594B	0	7

10.15 用 Python 语言编写的简单代码

如果计算时所使用的字长不必与使用神奇数字的环境所用的字长相等，那么神奇数字的计算过程就可极大地简化了。例如，对于无符号的情况来说，可使用 Python 语言直接求出 n_c，然后根据 10.9 节的（27）式与（26）式来计算 p 与 m。图 10.4 列出了这样一个函数。

```
def magicgu(nmax, d):
    nc = (nmax + 1 //d)*d - 1
    nbits = int(log(nmax, 2)) + 1
    for p in range(0, 2*nbits + 1):
        if 2**p > nc*(d - 1 - (2**p - 1)%d):
            m = (2**p + d - 1 - (2**p - 1)%d)//d
            return (m, p)
    print "Can't find p, something is wrong."
    sys.exit(1)
```

图 10.4 用 Python 语言计算无符号除法所用的神奇数字

调用此函数时，要传入被除数可能取到的最大值 nmax 以及除数 d。它返回一对整数：神奇数字 m 与移位量 p。要想计算被除数 x 除以 d 的商，可以先把 x 与 m 相乘，然后把（长度完整的）乘积右移 p 位。

此程序有两个方面比本章讲的其他算法更为通用：（1）使用此函数时要指定被除数有可能取到的最大值（nmax），而不指定被除数所具有的二进制位个数；（2）此程序可以处理任意大的被除数与除数（“大数”，bignum）。调用此函数时，要指定被除数的实际最大值，而不是容纳被除数的那些二进制位所能表示的理论最大值（那个数是其值为 2 的幂减 1 且不小于被除数的下一个整数），这样做的好处是有时找到的神奇数字较小。例如，假设被除数的实际最大值是 90，除数是 7。函数 magicgu 返回（37，8），这说明神奇数字是 37（此数能以 6 个二进制位来表示），而移位量是 8。但是，如果将被除数的理论最大值 127 传给该函数，则结果是（147，10），这样求出来的神奇数字 147 要用 8 个二进制位表示。

10.16　除数为常量的精确除法

"精确除法"（exact division）的意思是我们已经预先知道除法余数是 0 了[⊖]。虽说这类情况不常见，但有时的确会发生，例如在 C 语言中把两个指针相减。在 C 语言中，如果 p 与 q 是指针，那么只有当二者都指向同一数组中的对象时，$p-q$ 的结果才有明确含义，并且可以移植［H & S，sec. 7.6.2］。如果数组元素大小为 s，则目标代码会用 $(p-q)/s$ 来计算 $p-q$ 的差。

本节内容受［GM，sec. 9］启发而编写。

此方法适用于带符号与无符号的"精确除法"，它基于下述定理。

定理 MI　*若 a 与 m 互为质数，则存在满足 $1\leqslant\bar{a}<m$ 的整数 $\bar{a}$，使得下式[⊜]成立：*

$$a\bar{a}\equiv 1(\text{mod } m)。$$

也就是说，$\bar{a}$ 是 a 对 m 的模反元素[⊜]。此定理有很多种证明方法，［NZM，p.52］给出了三种。下面给出的这种证明方式只需要懂一些基本的同余（congruences）知识即可。

证明：我们要证明比该定理稍微通用一些的情况。如果 a 与 m 互为质数（这样的话，两数都不为零），那么当 x 取遍 m 种互不相同的余数值时，ax 除以 m 的余数也会取遍这 m 种余数值。比方说，假设 $a=3$ 且 $m=8$，当 x 取遍 0 至 7 之间的数时，ax 的值依次是 0、3、6、9、12、15、18、21，如果将这些 ax 值分别除以 8，则余数依次为 0、3、6、1、4、7、2、5。由此可见，0 至 7 这 8 个值都出现在了刚才这个余数序列中。

为了讲得更加通用些，我们假设以上论述不成立。那么，存在两个不同的整数，它们乘以 a 之后除以 m 所得余数相同，换句话说，存在符合 $x\not\equiv y(\text{mod } m)$ 的 x 与 y，使得

$$ax\equiv ay(\text{mod } m)$$

这样的话，也就存在整数 k，使得

$$ax-ay=km，\text{或者说}$$

$$a(x-y)=km$$

由于 a 与 m 没有公因数，所以 $x-y$ 必然是 m 的倍数，也就是说

$$x\equiv y(\text{mod } m)$$

这与 $x\not\equiv y$（mod m）这一假设相矛盾。

由于 ax 除以 m 的余数取遍了全部 m 个数，所以当 x 遍历各种 m 值时，必然会出现

⊖ 也就是俗称的"整除"。——译者注

⊜ $a\equiv b$（mod m）读作"a 同余于 b 模 m"、"a 与 b 对模 m 同余"、"a 与 b 关于模 m 同余"，意思是整数 a、b 除以正整数 m 后的余数相等。详情参见：http://zh.wikipedia.org/wiki/同余。——译者注

⊜ 原文为 $\bar{a}$ is a multiplicative inverse of a, modulo m，也称"$\bar{a}$ 是 a 关于 m 的模反元素"。"模反元素"的概念参见：http://zh.wikipedia.org/wiki/模反元素。原书作者在后文中把这一概念直呼为"multiplicative inverse"（乘法逆元素），这个词在本节中的含义比常用的"倒数"（对应英文单词也是"multiplicative inverse"）一词要广泛。译文将"模反元素"和原书作者所说的"乘法逆元素"视为同一概念。——译者注

当 x 取到某值时 ax 除以 m 余 1 的情况。

于是证明了对于 m 来说，只有 1 个这样的 x 值，能使 $ax \equiv 1 \pmod{m}$ 成立，换句话说，除去自身与 m 的各种倍数相加之和以外，“模反元素”是唯一的[⊖]。这也证明了只有一个大于 0 且小于 m 的整数 x，能够使 $ax \equiv b \pmod{m}$ 成立，其中 b 为任意整数。 ■

举例来说，考虑 $m=16$ 的情况。由于 $3 \cdot 11 = 33 \equiv 1 (\bmod 16)$，所以 $\overline{3}=11$。因为 $3 \cdot (-5) = -15 \equiv 1 (\bmod 16)$，所以也可以说 $\overline{3}=-5$。同理，因为 $(-3) \cdot 5 = -15 \equiv 1 (\bmod 16)$，所以还可以说 $-\overline{3}=5$。

这一规律很重要，因为它表明，此概念既适用于带符号数，也适用于无符号数。如果想在 4 位计算机中计算无符号整数，那么就认定 $\overline{3}=11$。若要做带符号整数运算，则定义 $-\overline{3}=5$。由于 11 与 -5 用“2 补码”表示出来是一样的（因为二者之差为 16），所以可用内容相同的字组来充当无符号与带符号这两种情况下的“乘法逆元素”（multiplicative inverse）。

此定理可直接用来在 W 位计算机上执行除数 d 为奇数的除法（带符号和无符号均可）。由于任何奇数都与 2^W 互质，所以根据早前的定理可得：如果 d 为奇数，则有整数 $-\overline{d}$（在 0 至 2^W-1 或 -2^{W-1} 至 $2^{W-1}-1$ 这个范围内，此数唯一）使得下式成立：

$$d\overline{d} \equiv 1 (\bmod 2^W)$$

因此，对于任意整数 n 来说，如果其为 d 的倍数，则有

$$\frac{n}{d} \equiv \frac{n}{d}(d\overline{d}) \equiv n\overline{d} (\bmod 2^W)$$

换句话说，要计算 n/d，可以先把 n 乘以 $\overline{d}$，然后只取乘积最右侧 W 位即可。

若除数 d 为偶数，则将 d 表示为 $d_o \cdot 2^k$，其中 d_o 为奇数且 $k \geqslant 1$。这样一来，将 n 右移 k 位（也就是移走尾部的“0”），然后再乘以 $\overline{d_o}$ 即可（也可以先做乘法后移位）。

下面列出计算 n 除以 7 所用的代码，其中 n 是 7 的倍数。不论是带符号除法还是无符号除法，代码运算结果都正确。

```
li    M,0xB6DB6DB7  Mult. inverse, (5*2**32 + 1)/7.
mul   q,M,n         q = n/7.
```

10.16.1 用欧几里得算法计算乘法逆元素

如何计算乘法逆元素呢？标准方法是采用“扩展版欧几里得算法”（extended Euclidean algorithm）。由于此算法与本书内容有关，所以将会在下面简述，读者如欲探知详情，可参考［NZM，p.13］与［Knu2，4.5.2］。

给定奇除数 d，需要求解下式中的 x：

$$dx \equiv 1 (\bmod m)$$

⊖ 意思是，假如 $\bar{a}$ 是 a 的模反元素，则 $\bar{a}+1m$、$\bar{a}+2m$、$\bar{a}+3m$、… 这些数与 a 的乘积除以 m 后也余 1。——译者注

在本节所讲的应用场景中，$m=2^W$ 而 W 为计算机字长。如果能解出下列等式中的整数 x，y（正数、负数、0 均可），那么也就找到乘法逆元素了。

$$dx+my=1$$

为了求解上式，第一步是为 d 加上一定倍数的 m，使之变成正数。（用 d 与用 $d+km$ 算出来的乘法逆元素相同。）第二步，写出下面两个等式（其中 d 与 m 均大于 0）：

$$d(-1)+m(1)=m-d \quad \text{(i)}$$

$$d(1)+m(0)=d \quad \text{(ii)}$$

若 $d=1$，则运算完毕，因为由（ii）式已经知道 $x=1$ 了。否则，就计算

$$q=\left\lfloor \frac{m-d}{d} \right\rfloor$$

第三步，将等式（ii）乘以 q，并用等式（i）减去它。于是可得

$$d(-1-q)+m(1)=m-d-qd=\mathrm{rem}(m-d,d)$$

由于刚才只是把一个等式乘以常数，并将其从另一个等式中减去，所以上面得出的这个等式成立。要是 $\mathrm{rem}(m-d,d)=1$，那么运算完毕，最后这个等式就是我们想求的解，此时 $x=-1-q$。

运用最后两个等式，求得第 4 个等式，反复执行此过程，直至等式右侧为 1。此时将最后一个式子中 d 的乘数与 m 取模，余数就是 d 的“乘法逆元素”。

顺便说一下，如果 $m-d<d$，那么首次求出的商就是 0，而第 3 个方程式也就和（i）相同了，这样的话，第二次求出的商肯定不是 0。此外，大多数教科书都把（i）写成

$$d(0)+m(1)=m$$

但是在本节所讲的应用场景中，计算机无法表示 $m=2^W$ 这个数。

最好用范例演示一下这个过程，假设 $m=256$ 而 $d=7$。计算过程如下。看到第 3 行的时候，注意 d 的乘数是根据 $q=\lfloor 249/7 \rfloor=35$ 算出的。

```
 7(-1) + 256( 1) = 249
 7( 1) + 256( 0) = 7
7(-36) + 256( 1) = 4
7( 37) + 256(-1) = 3
7(-73) + 256( 2) = 1
```

因此，7 对 256 的模反元素是 -73，如果将其表示为 0 至 255 之间的数，那就是 183。验算：$7\cdot 183=1281\equiv 1 \pmod{256}$。

从第 3 行开始，等式右边那一栏中的整数等于：从它往上数第二行的对应整数，除以它上面那一行的对应整数所得之余数，于是，这些数就构成了严格递减（strictly decreasing）的非负整数序列。因此，这个序列必定以 0 结尾（如果把刚才演示的运算步骤再往下执行一步的话，就会出现这种情况）。而且，0 之前的那个数必定是 1，接下来要解释其原因。假设序列最后两个数是 b 和 0，且 $b\neq 1$。那么，由于 b 之后的余数是 0，所以 b 之前的那个数必定是 b 的倍数，设其为 k_1b。由于 k_1b 之后的余数是 b，所以它之前的整数必定是 k_1k_2b+b 的形式。回溯整个序列：每个数字都必须是 b 的倍数，包括头两个

（按照上面的例子来说，就是249和7所在的位置）在内。因为已经知道前两个整数 $m-d$ 与 d 互质了，所以这是不可能的。

这就以非严格的形式证明了上述过程必将终止，且最后一个式子的右侧是1，因而，此方法能找到 d 的乘法逆元素。

要在计算机上实现此算法，首先得注意，如果 $d<0$，应该为其加上 2^W。由于计算机执行的是"2补码"算术，所以实际上无需任何操作，不论应用场景如何解读 d，我们只把它当成无符号数就好。

计算 q 则必须使用无符号除法。

注意到算法可以采用"模 m"的方式[⊖]计算，因为这并不改变每个式子右侧那一栏的值（这些值无论如何都在0与 $m-1$ 之间）。这一点很重要，因为它意味着算法只需使用计算机的"模 2^W 式无符号算术"（modulo-2^W unsigned arithmetic）执行"单精度计算"（single precision）[⊜]即可。

早前方程式表格中的大部分数都不必表示出来。因为在求解 $dx+my=1$ 时不关心 y 的值，所以与256的倍数的那一栏无需写出。第一栏中的 d 也不用写出。以最简形式将上例中的计算过程表述如下：

```
255  249
  1    7
220    4
 37    3
183    1
```

图10.5列出了执行此计算所用的C语言代码。

```
unsigned mulinv(unsigned d) {           // d must be odd.
   unsigned x1, v1, x2, v2, x3, v3, q;

   x1 = 0xFFFFFFFF;      v1 = -d;
   x2 = 1;               v2 = d;
   while (v2 > 1) {
      q = v1/v2;
      x3 = x1 - q*x2;    v3 = v1 - q*v2;
      x1 = x2;           v1 = v2;
      x2 = x3;           v2 = v3;
   }
   return x2;
}
```

图10.5 用欧几里得算法求"模 2^{32}"的乘法逆元素

while 循环的继续条件用的是"(v2>1)"，而不是更为自然的"(v2 != 1)"，因为如果用后者，那么程序接受了值为偶数的参数后，循环就停不下来了。在调用者误用了本程序后，最好别让它一直循环下去。（若参数 d 是偶数，则 v2 永远不可能是1，不过它会

⊖ 意思是可以把算出来的值和 m 取模，将余数当成运算结果。——译者注

⊜ 这里是比喻"不求完整结果，只求其除以 2^W 的余数"这一行为，与"单精度浮点数"一词中的"单精度"含义不同。——译者注

变成 0。）

如果用偶数做参数，程序算出来的会是什么呢？按照现在写的代码来看，程序会求出满足 $dx \equiv 0 \pmod{2^{32}}$ 的 x 值，这恐怕没有用。然而，只需稍加修改，把循环继续条件变成“(v2 != 0)”，并将返回值由 x2 改为 x1，这样就能算出满足 $dx \equiv g \pmod{2^{32}}$ 的 x 了，其中 g 是 d 和 2^{32} 的最大公约数，也就是说，g 是 2 的幂，它能整除 d，且其值最大。修改后的程序在 d 为奇数时仍能算出 d 的乘法逆元素，但是要比未修改的程序多执行一次循环。

从循环执行次数（也就是执行除法操作的次数）来看，当 d 为小于 20 的奇数时，上述程序最多循环 3 次，平均循环 1.7 次。若 d 在 1000 左右，则最多需循环 11 次，平均约 6 次。

10.16.2 用牛顿法计算乘法逆元素

众所周知，对于实数来说，可以反复套用下式以计算 $1/d$ ($d \neq 0$)，只要初始值 x_0 足够接近 $1/d$，那么算出来的值就越来越精确：

$$x_{n+1} = x_n(2 - dx_n) \tag{31}$$

每次计算完一次，精确数位的个数就比原来翻一番。

然而很多人不知道的是，对于模 2 的任意次幂来说，用这个公式可以找出乘法逆元素。例如想求出 3 对于模 256 的乘法逆元素，可以用 $x_0 = 1$ 作初始值（任意奇数都可充当初始值）。于是可得

$$\begin{aligned}
x_1 &= 1\ (2 - 3 \cdot 1) = -1 \\
x_2 &= -1\ (2 - 3\ (-1)) = -5 \\
x_3 &= -5\ (2 - 3\ (-5)) = -85 \\
x_4 &= -85\ (2 - 3\ (-85)) = -21\ 845 \equiv -85 \pmod{256}
\end{aligned}$$

反复执行迭代，最终会达到一个定点，使得每次算出的数字和 256 取模后的余数总相同，所以说 −85 或者 171 就是 3 对于模 256 的乘法逆元素。上述过程中所有计算都可按照“模 256”的方式来做。

为何能用牛顿法求出乘法逆元素呢？因为假如 x_n 满足

$$dx_n \equiv 1 \pmod{m}$$

且 x_{n+1} 是按照 (31) 式算出来的，那么就有

$$dx_{n+1} \equiv 1 \pmod{m^2}$$

为了推导出上述结论，设 $dx_n = 1 + km$。于是有

$$\begin{aligned}
dx_{n+1} &= dx_n(2 - dx_n) \\
&= (1 + km)(2 - (1 + km)) \\
&= (1 + km)(1 - km) \\
&= 1 - k^2 m^2 \\
&\equiv 1 \pmod{m^2}
\end{aligned}$$

在本节的应用中，m 是 2 的幂，比如可以把它写成 2^N。在这种情况下，如果

$$dx_n \equiv 1(\bmod 2^N),\text{那么}$$
$$dx_{n+1} \equiv 1(\bmod 2^{2N})$$

从某种意义上说，如果 x_n 是 $\overline{d}$ 的近似值，那么每次执行完（31）式之后，“精确”的二进制位个数就比之前提升大约一倍。

对于模 8 来说，任何奇数 d 的乘法逆元素就是 d 本身。因此，直接用 $x_0=d$ 来猜测 $\overline{d}$ 的初始值就显得比较合理了。于是，反复运用（31）式求出 x_1，x_2，… 的值：

$$dx_1 \equiv 1(\bmod 2^6),$$
$$dx_2 \equiv 1(\bmod 2^{12}),$$
$$dx_3 \equiv 1(\bmod 2^{24}),$$
$$dx_4 \equiv 1(\bmod 2^{48}),\text{等等。}$$

因此，对于模 2^{32} 来说，只需 4 次迭代就可找到乘法逆元素了（如果 $x \equiv 1(\bmod 2^{48})$，那么当 n 小于等于 48 时，就有 $x \equiv 1(\bmod 2^n)$）。此思路可归结为图 10.6 中的 C 语言代码，其中所有运算都按照“模 2^{32}”的方式来做。

在 d 可能取到的值中，有一半取值会令程序执行 4.5 次迭代，或者说执行 9 次乘法。而对于另外一半取值（也就是 xn 的初始值已经猜对了最右侧 4 个二进制位的那些情况，即 $d^2 \equiv 1(\bmod 16)$）来说，程序执行乘法不会超过 7 次，通常就是 7 次。因此，该程序平均要执行约 8 次乘法。

有一种变体算法，不管 d 为何值，一律执行 4 次循环，不过为了去除循环控制指令，可能会把这 4 轮循环依次排开（string out），这样的话一共执行 8 次乘法。还有一种变体：在估算初始值 x_0 时，设法猜对答案最右侧的 4 个二进制位（也就是找到满足 $dx_0 \equiv 1(\bmod 16)$ 的 x_0）。然后只需 3 次循环迭代即可。下列二式都可用来估算初始值：

$$\boldsymbol{x_0 \leftarrow d + 2((d+1)\,\&\,4)},$$
$$\boldsymbol{x_0 \leftarrow d^2 + d - 1}$$

```
unsigned mulinv(unsigned d) {        // d must be odd.
   unsigned xn, t;

   xn = d;
loop: t = d*xn;
      if (t == 1) return xn;
      xn = xn*(2 - t);
      goto loop;
}
```

图 10.6　用“牛顿法”求对于模 2^{32} 的乘法逆元素

此处以左移位来实现乘以 2，所有计算都按“模 2^{32}”的方式执行，并忽略溢出。由于第二个公式用了一次乘法，所以只比第一个公式省了一条指令。

如果要将此算法集成到编译器中，那么刚才讨论的执行时间问题就完全不重要了。因为在那种场合下，很少用到此例程，所以应该尽量缩减代码所占空间。不过，在某些

应用场景中，可能需要迅速算出乘法逆元素。

此处所述的牛顿法（Newton method）仅适用于如下情况：（1）模是某数 a 的整数幂，并且（2）我们已经知道 d 对于模 a 的乘法逆元素了。因为任何奇数对于模 2 的乘法逆元素都必然是 1，所以此方法在 $a=2$ 时效果尤佳。

10.16.3 乘法逆元素示例

本节最后，我们将某些乘法逆元素列在表 10.3 中。

表 10.3 乘法逆元素示例

d	$\bar{d}$		
（十进制）	模 16 之后的十进制值	模 2^{32} 之后的十六进制值	模 2^{64} 之后的十六进制值
−7	−7	49249249	9249249249249249
−5	3	33333333	3333333333333333
−3	5	55555555	5555555555555555
−1	−1	FFFFFFFF	FFFFFFFFFFFFFFFF
1	1	1	1
3	11	AAAAAAAB	AAAAAAAAAAAAAAAB
5	13	CCCCCCCD	CCCCCCCCCCCCCCCD
7	7	B6DB6DB7	6DB6DB6DB6DB6DB7
9	9	38E38E39	8E38E38E38E38E39
11	3	BA2E8BA3	2E8BA2E8BA2E8BA3
13	5	C4EC4EC5	4EC5EC5EC5EC5EC5
15	15	EEEEEEEE	EEEEEEEEEEEEEEEF
25		C28F5C29	8F5C28F5C28F5C29
125		26E978D5	1CAC083126E978D5
625		3AFB7E91	D288CE703AFB7E91

读者也许注意到了，某些情况下（当 $d=3$，5，9，11 时），d 的乘法逆元素与除数为 d 的无符号除法所用之神奇数字（参见 10.14 节）相同。这有几分巧合。对于这些除数来说，神奇数字 M 就等于乘数 m，而这些乘数都具备 $(2^p+1)/d$ 的形式，其中 $p \geqslant 32$。在发生此现象时，下式成立：

$$Md=\left(\frac{2^p+1}{d}\right)d \equiv 1 \pmod{2^{32}}$$

所以 $M \equiv \bar{d} \pmod{2^{32}}$。

10.17 检测除以常数后是否余 0

除数 d 的乘法逆元素可用来检测某数除以 d 后是否余 0［GM］。

10.17.1 无符号除法

首先考虑除数 d 为奇数的无符号除法。用 $\bar{d}$ 表示 d 的乘法逆元素。由于 $d\bar{d}\equiv 1 \pmod{2^W}$，其中 W 是以位元为单位的计算机字长，$\bar{d}$ 也是奇数。由此可知，$\bar{d}$ 与 2 互质，而且由前一节证明定理 MI 的过程可知，当 n 取遍除数为 2^W 的除法可能产生的 2^W 种余数时，$n\bar{d}$ 也会取遍这互不相同的 2^W 个余数。

上一节已经证明，若 n 为 d 的倍数，则

$$\frac{n}{d}\equiv n\bar{d} \pmod{2^W}$$

也就是说，当 $n=0$，d，$2d$，…，$\lfloor(2^W-1)/d\rfloor d$ 时，$n\bar{d}\equiv 0$，1，2，…，$\lfloor(2^W-1)/d\rfloor \pmod{2^W}$。因此，如果 n 不是 d 的倍数，那么将 $n\bar{d}$ 和 2^W 取模，把其值缩减到 0 至 2^W-1 之间，这个值就必将超过 $\lfloor(2^W-1)/d\rfloor$。

利用此论述可判断除数是否为 0。例如，要检测整数 n 是否为 25 的倍数，可以把 n 与 $\overline{25}$ 相乘，然后比较乘积右侧 W 位是否等于 $\lfloor(2^W-1)/25\rfloor$。用基本 RISC 指令可以这样写：

```
li      M,0xC28F5C29  Load mult. inverse of 25.
mul     q,M,n         q = right half of M*n.
li      c,0x0A3D70A3  c = floor((2**32-1)/25).
cmpleu  t,q,c         Compare q and c, and branch
bt      t,is_mult     if n is a multiple of 25.
```

为了将此方法扩展至偶除数，设 $d=d_0\cdot 2^k$，其中 d_0 是奇数且 $k\geqslant 1$。当且仅当某数能被 d_0 与 2^k 整除时，它才能为 d 所整除，而 n 与 $n\bar{d}_0$ 的后缀 0 个数相同（$\bar{d}_0$ 是奇数），基于上述二原因，可按如下方式检测 n 是否为 d 的倍数：

设 $q=\mathrm{mod}(n\bar{d}_0,\ 2^W)$；

判断 $q\leqslant\lfloor(2^W-1)/\bar{d}_0\rfloor$ 是否成立，并判断 q 尾部是否有 k 个或更多个“0”。

使用 mod 函数是为了将 $n\bar{d}_0$ 调整到 $[0,\ 2^W-1]$ 这个区间内。

直接实现上述算法需要两套测试与条件分支指令，然而若是计算机支持循环移位指令，那么只需 1 套比较与分支指令（compare-branch）就可以很高效地实现出来。改进后的算法基于如下定理，其中 $a\overset{rot}{\gg}k$ 表示将计算机字组 a 循环右移 k 位（$0\leqslant k\leqslant 32$）。

定理 ZRU 当且仅当 $\boldsymbol{x}\overset{rot}{\gg}k\overset{u}{\leqslant}\lfloor\boldsymbol{a}/2^k\rfloor$ 时，$\boldsymbol{x}\overset{u}{\leqslant}\boldsymbol{a}$ 成立，同时 $\boldsymbol{x}$ 的结尾至少有 k 个“0”位元。

证明（假定使用 32 位计算机）：假设 $\boldsymbol{x}\overset{u}{\leqslant}\boldsymbol{a}$ 且 $\boldsymbol{x}$ 结尾至少有 k 个“0”位元。那么，由于 $\boldsymbol{x}\overset{u}{\leqslant}\boldsymbol{a}$，所以 $\lfloor\boldsymbol{x}/2^k\rfloor\overset{u}{\leqslant}\lfloor\boldsymbol{a}/2^k\rfloor$。而 $\lfloor\boldsymbol{x}/2^k\rfloor=\boldsymbol{x}\overset{rot}{\gg}\boldsymbol{k}$，因此，$\boldsymbol{x}\overset{rot}{\gg}k\overset{u}{\leqslant}\lfloor\boldsymbol{a}/2^k\rfloor$。如果 $\boldsymbol{x}$ 的结尾不足 k 个“0”位元，那么 $\boldsymbol{x}\overset{rot}{\gg}k$ 的开头也不足 k 个“0”位元，然而 $\lfloor\boldsymbol{a}/2^k\rfloor$ 的开头却有 k 个“0”位元，所以 $\boldsymbol{x}\overset{rot}{\gg}k\overset{u}{>}\lfloor\boldsymbol{a}/2^k\rfloor$。最后，如果 $\boldsymbol{x}\overset{u}{>}\boldsymbol{a}$ 而 $\boldsymbol{x}$ 的结尾又至少有 k 个“0”位元，那么由 $\boldsymbol{x}$ 的前 $32-k$ 个位元所构成的整数必然超过了由 $\boldsymbol{a}$ 的前 $32-k$ 个位元所构成的整数，

因此$\lfloor x/2^k \rfloor \overset{u}{>} \lfloor a/2^k \rfloor$。 ■

根据此定理，当 n 与 d（d 大于等于 1）为无符号整数，且 $d=d_0 \cdot 2^k$之中的 d_0为奇数时，可用下述方法检测 n 是否为 d 的倍数：

$$\boxed{\begin{aligned}&\boldsymbol{q} \leftarrow \mathrm{mod}(\boldsymbol{n}\,\overline{\boldsymbol{d}}_0, \mathbf{2}^W)\\ &\quad \boldsymbol{q} \overset{rot}{\gg} \boldsymbol{k} \overset{u}{\leqslant} \lfloor (\mathbf{2}^W - \mathbf{1})/\boldsymbol{d} \rfloor\end{aligned}}$$

在推导上述方法的过程中用到了这个式子：$\lfloor\lfloor(2^W-1)/d_0\rfloor/2^k\rfloor=\lfloor(2^W-1)/(d_0 \cdot 2^k)\rfloor=\lfloor(2^W-1)/d\rfloor$。

举个例子，下列代码可判断无符号整数 n 是不是 100 的倍数：

```
li      M,0xC28F5C29  Load mult. inverse of 25.
mul     q,M,n         q = right half of M*n.
shrri   q,q,2         Rotate right two positions.
li      c,0x028F5C28  c = floor((2**32-1)/100).
cmpleu  t,q,c         Compare q and c, and branch
bt      t,is_mult     if n is a multiple of 100.
```

10.17.2 除数大于等于 2 的带符号除法

对于带符号除法来说，上节已经证明，若 n 为 d 的倍数且 d 是奇数，那么

$$\frac{n}{d} \equiv n\,\overline{d} \pmod{2^W}$$

因此，当 $n=\lceil -2^{W-1}/d \rceil \cdot d,\ldots,-d,0,d,\ldots,\lfloor(2^{W-1}-1)/d\rfloor \cdot d$ 时，$n\overline{d} \equiv \lceil -2^{W-1}/d \rceil,\ldots,-1,0,1,\ldots,\lfloor(2^{W-1}-1)/d\rfloor \pmod{2^W}$。此外，由于 d 与 2^W互质，所以当 n 取遍除以 2^W后可能产生的 2^W种余数时，$n\overline{d}$也会取遍这 2^W个余数。因此，当且仅当下式成立时，n 才是 d 的倍数：

$$\lceil -2^{W-1}/d \rceil \leqslant \mathrm{mod}(n\,\overline{d}, 2^W) \leqslant \lfloor(2^{W-1}-1)/d\rfloor$$

其中使用 mod 函数是为了将 $n\overline{d}$的值调整到 $[-2^{W-1}, 2^{W-1}-1]$ 这个区间内。

上述算式看上去还能再简化一下。由于 d 是奇数，而且本节假设它是不等于 1 的正数，所以它无法整除 2^{W-1}，于是可得

$$\lceil -2^{W-1}/d \rceil = \lceil(-2^{W-1}+1)d\rceil = -\lfloor(2^{W-1}-1)/d\rfloor$$

因此，对于带符号数来说，可按下述方法判断 n 是否为 d 的倍数（$d=d_o \cdot 2^k$且 d_o为奇数）：

设 $q=\mathrm{mod}\ (n\overline{d}_o,\ 2^W)$；

判断$-\lfloor(2^{W-1}-1)/d_o\rfloor \leqslant q \leqslant \lfloor(2^{W-1}-1)/d_o\rfloor$是否成立，并判断 q 尾部是否有 k 个或更多个“0”位元。

表面上看，似乎需要 3 套测试与分支指令。然而，与无符号数的情况类似，也可使用如下定理将其简化成一套比较与分支（compare-branch）指令。

定理 ZRS 若 $\boldsymbol{a} \geqslant 0$，则下列断言（assertion）等价：

(1) $-a \leqslant x \leqslant a$，且 x 尾部至少有 k 个值为“0”的位元

(2) abs $(x) \overset{rot}{\gg} k \overset{u}{\leqslant} \lfloor a/2^k \rfloor$

(3) $x + a' \overset{rot}{\gg} k \overset{u}{\leqslant} \lfloor 2a'/2^k \rfloor$

其中 a' 是把 a 最右侧 k 个位元设为“0”而得的值（也就是说，$a' = a \,\&\, -2^k$）。

证明（假定在32位计算机上）：为了证明(1)与(2)等价，首先已知 $-a \leqslant x \leqslant a$ 与 $\text{abs}(x) \leqslant a$ 显然等价。然后，由于不等式 $\text{abs}(x) \leqslant a$ 的两端非负，所以运用定理ZRU可证其与(2)等价。

为证明(1)与(3)等价，我们注意到，若将 a 替换为 a'，则断言(1)与其自身等价。然后，根据4.1节的“边界检测定理”，替换后的断言又等价于：

$$x + a' \overset{u}{\leqslant} 2a'$$

由于当且仅当 x 尾部至少有 k 个“0”位元时，$x + a'$ 的尾部才能有至少 k 个“0”位元，所以根据定理ZRU可知此断言与(3)等价。 ■

根据定理中的断言(3)，当 n 与 d 为带符号整数（d 大于等于2），且 $d = d_o \cdot 2^k$ 中的 d_o 为奇数时，可用下述方法判定 n 是否为 d 的倍数。

$$\boxed{\begin{aligned} & q \leftarrow \text{mod}(n\bar{d}_0, 2^W) \\ & \qquad a' \leftarrow \lfloor (2^{W-1} - 1/d_0) \rfloor \& -2^k; \\ & \qquad q + a' \overset{rot}{\gg} k \overset{u}{\leqslant} \left\lfloor (2a')/2^k \right\rfloor 。 \end{aligned}}$$

（由于 d 为常数，所以 a' 可在编译时计算。）

举个例子，下列代码可判断带符号整数 n 是否为100的倍数。注意到常量 $\lfloor 2a'/2^k \rfloor$ 总是能通过将 a' 右移 $k-1$ 位而得出，所以在计算比较指令的操作数[㊀]时，可以节省一条计算指令或是一条内存加载指令。

```
li     M,0xC28F5C29   Load mult. inverse of 25.
mul    q,M,n          q = right half of M*n.
li     c,0x051EB850   c = floor((2**31 - 1)/25) & -4.
add    q,q,c          Add c.
shrri  q,q,2          Rotate right two positions.
shri   c,c,1          Compute const. for comparison.
cmpleu t,q,c          Compare q and c, and
bt     t,is_mult      branch if n is a mult. of 100.
```

10.18 不使用 Multiply High 指令的除法算法

本节要再讲一些除数为常数的除法算法，这些算法不使用“multiply high”指令[㊁]，

㊀ 指的是代码中 cmpleu 指令的操作数 c。——译者注

㊁ 指的是“multiply high signed”（简称 mulhs，求带符号乘法乘积的高权重部分）与“multiply high unsigned”（简称 mulhu，求无符号乘法乘积的高权重部分）。——译者注

也不使用以双字存放乘积的乘法指令。大家将看到如何把除数为常数的除法转换为一系列移位及加法指令，以及如何用移位、加法、乘法指令将其编为更加精简的代码。

10.18.1　无符号除法

对于要讲的这些算法来说，无符号除法的情况要比带符号除法简单，所以先看无符号除法。一种办法是，在技巧上借鉴使用“multiply high”指令的那种算法，但是在实现上采用 8.2 节的图 8.2 来执行“multiply high”操作。图 10.7 演示了如何以此思路计算除数为 3 的无符号除法。这段代码将 10.8.1 节的算法与图 8.2 结合起来，并把后者所用的“int”换成了“unsigned”。此代码共需 15 条指令，其中 4 条是乘法。乘法指令使用的常数都比较大，若将其改写成移位及加法操作，则需要很多条指令。带符号除法所用的代码与之非常接近。此算法不是特别好，所以不再深入讨论了。

另一个方法［GLS1］是预先算好除数的倒数，然后用一系列右移及加法指令计算倒数与被除数的乘积，这样就能估算出商了。这仅仅是个近似值，因为除数的倒数未必能用 32 个二进制位精确地表示出来（假设除数不一定是 2 的幂），而且右移指令还会丢掉被除数里的某些位元。接下来，根据估算出的商求得余数，并将此余数除以除数，其结果用作修正值，把修正值加到估算出的商里，就得到了准确的商。与除数（或余数的几倍）相比，余数一般比较小，所以通常有种简单的方法能算出修正值，无需使用除法指令。

```
unsigned divu3(unsigned n) {
   unsigned n0, n1, w0, w1, w2, t, q;

   n0 = n & 0xFFFF;
   n1 = n >> 16;
   w0 = n0*0xAAAB;
   t  = n1*0xAAAB + (w0 >> 16);
   w1 = t & 0xFFFF;
   w2 = t >> 16;
   w1 = n0*0xAAAA + w1;
   q = n1*0xAAAA + w2 + (w1 >> 16);
   return q >> 1;
}
```

图 10.7　用模拟出来的“求无符号乘法高权重积”(multiply high unsigned) 操作计算除数为 3 的无符号除法

为了演示此方法，思考除数为 3 的情况，也就是计算 $\lfloor n/3 \rfloor$，其中 $0 \leqslant n < 2^{32}$。3 的倒数用二进制表示大概是

0.0101 0101 0101 0101 0101 0101 0101 0101

可以用下式估算出此倒数和 n 的乘积：

$$\boldsymbol{q} \leftarrow (\boldsymbol{n} \overset{u}{\gg} 2) + (\boldsymbol{n} \overset{u}{\gg} 4) + (\boldsymbol{n} \overset{u}{\gg} 6) + \ldots + (\boldsymbol{n} \overset{u}{\gg} 30) \tag{32}$$

（共需 29 条指令，倒数中最后的那个“1”可以忽略，因为它相当于把 $\boldsymbol{n} \overset{u}{\gg} 32$ 这一项累加进去，而这一项显然是 0）。不过，由于倒数中反复出现简单的“0”、“1”组合，所以可改用另一种既快又准的方法来计算乘积（需要 9 条指令）。

$$\begin{aligned}
&\boldsymbol{q} \leftarrow (\boldsymbol{n} \overset{u}{\gg} 2) + (\boldsymbol{n} \overset{u}{\gg} 4) \\
&\boldsymbol{q} \leftarrow \boldsymbol{q} + (\boldsymbol{q} \overset{u}{\gg} 4) \\
&\boldsymbol{q} \leftarrow \boldsymbol{q} + (\boldsymbol{q} \overset{u}{\gg} 8) \\
&\boldsymbol{q} \leftarrow \boldsymbol{q} + (\boldsymbol{q} \overset{u}{\gg} 16)
\end{aligned} \tag{1}$$

现在比较一下这两种方法的精确程度：假设 n 的所有二进制位全是“1”，那么（32）式中的每一项所移出去的位分别是什么呢？第一项移出去两个“1”，接下来那项移出去 4 个“1”，以此类推。由于共有 16 项（算上忽略的那项），所以移位操作产生的误差几乎接近 16。再加上因倒数截取成 32 位而引入的误差，总计最大误差值是 16。

如果用方式（1）来计算，那么每次右移操作会在最低有效位上产生近乎 1 的误差。然而这种方法只用了 5 次移位操作。它们加起来的误差很接近 5，再加上因倒数截取成 32 位而引入的误差，总计最大误差值是 5。

在估算出商数 q 之后，根据下式算出余数 r：

$$r \leftarrow n - q * 3$$

由于 q 总是比真实的商小，所以余数不可能为负。为了找寻 $r \overset{u}{\div} 3$ 的最简算法，我们需知道 r 最大可能是多少。一般来说，如果除数是 d，而估算的商 q 比真实商小 k，那么余数就在 $k * d$ 至 $k * d + d - 1$ 这个范围内。（上限是保守估计，实际也许达不到。）采用方式（1）算出来的 q 最多比真实商小 5，因此，余数最大就是 $5 * 3 + 2 = 17$。通过实验发现，实际上余数最大为 15。这样的话，为了修正除法结果，必须准确计算出

$$r \overset{u}{\div} 3,\ 0 \leqslant r \leqslant 15$$

和寄存器所能容纳的最大值相比，r 很小，所以想求出上式的值，可以给 r 乘上一个接近 1/3 的数，而这个数具备 a/b 的形式，其中 b 是 2 的幂。这样一来就比较好算了，因为除法变成了简单的移位操作。a/b 的值要略大于 1/3，以便移位后的值能与“向 0 取整式除法”相符。下面是一系列接近 1/3 的值：

1/2，2/4，3/8，6/16，11/32，22/64，43/128，86/256，171/512，342/1024，…

在上述序列中，小一些的分数通常算起来比较方便，所以我们选用分子、分母都最小并且能求得正确结果的那个分数，在本例中，这个数就是 11/32。因此，最后可按下式算出准确的商：

$$q \leftarrow q + (11 * r \overset{u}{\gg} 5)$$

整个计算过程要用到两次操作数较小的乘法（分别是 3 和 11），它们可改用移位及加法来实现。

图 10.8 列出了完整的 C 语言代码。据此可知，此算法需要 14 条指令，其中两条是乘法。如果用移位及加法操作实现乘法，那么需要 18 条基本指令。然而若要刻意避免乘法指令，则可用另外两种方式改写返回语句，每种改写后的算法都需要 17 条基本指令。第 2 种改写方式略微提升了指令级并发执行能力，不过实际上发挥不了多大效果。

有种办法能更准确地估算商数。把第一行可执行代码改写成

```
q = (n >> 1) + (n >> 3);
```

（这行代码算出来的 q 会是原来那种算法的 2 倍多，然而其精确度高了一位），然后在赋值给 r 的语句前插入这行代码：

```
q = q >> 1;
```

这样修改后，余数最多是 9。但是，将 *r* 最大值限定在 9 之后，编不出更好的代码来计算 $r \overset{u}{\div} 3$ 了，和原来那种 *r* 最大为 15 的算法一样，都需要 4 条基本指令。这样看来，使用此思路反而增加了一条指令[⊖]。然而对大部分除数来说，使用这种思路确实都能改进代码，所以要在这里提一下。

```
unsigned divu3(unsigned n) {
   unsigned q, r;

   q = (n >> 2) + (n >> 4);        // q = n*0.0101 (approx).
   q = q + (q >> 4);               // q = n*0.01010101.
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*3;                    // 0 <= r <= 15.
   return q + (11*r >> 5);         // Returning q + r/3.
// return q + (5*(r + 1) >> 4);            // Alternative 1.
// return q + ((r + 5 + (r << 2)) >> 4);// Alternative 2.
}
```

图 10.8　除数为 3 的无符号除法

图 10.9 分别描述了用这两种思路如何计算除数为 5 的除法。5 的倒数写成二进制是：

0.0011 0011 0011 0011 0011 0011 0011 0011

与除以 3 时的情况类似，由于反复出现简单的“0”、“1”组合，所以能够高效而准确地估算出商数。左侧代码估算出来的商离正确值最多差 5，而余数离正确值最多差 25。右侧代码估算出来的商比左侧多精确两个位元，离正确值最多差 2。此时余数离正确值最多差 10。因为余数的最大误差比左侧算法小，所以在将余数与 1/5 相乘时可用 7/32 来估算乘积，而不像左侧代码那样要使用 13/64 来估算，这样一来，若用移位与加法指令实现乘法，则此算法可略微提高程序效率。从指令数量上看，左侧代码需要 14 条指令，其中两条是乘法，若改用基本指令来实现乘法，则需 18 条；右侧代码需要 15 条指令，其中两条是乘法，若改用基本指令来实现乘法，则需 17 条。只有在计算机支持比较谓词指令（comparison predicate instruction）时，才能用另外一种方式改写返回语句。改写后的代码没有节省指令，不过略微提高了并发执行能力。

```
unsigned divu5a(unsigned n) {
   unsigned q, r;

   q = (n >> 3) + (n >> 4);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*5;
   return q + (13*r >> 6);
}
```

```
unsigned divu5b(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*5;
   return q + (7*r >> 5);
// return q + (r>4) + (r>9);
}
```

图 10.9　除数为 5 的无符号除法

⊖ 当指计算 q = q≫1 所使用的移位指令。——译者注

对于除数为 6 的情况，可以使用除以 3 时所用的代码，然后再右移一位。然而，若使用下列二进制近似值来直接计算，则可再省一条乘法指令：

$$4/6 \approx 0.1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010$$

代码列在图 10.10 中。左侧算法先把被除数和一个近似 1/6 的数相乘，估算出商，然后再将余数乘以 11/64，以此修正刚才的商。右侧代码则把被除数和一个近似 4/6 的值相乘，这样做的好处在于，估算出来的余数最多离正确值差 1。这种算法使用的修正代码也变简单了：只需在 $r \geqslant 6$ 时加 1 即可。如果计算机支持比较谓词指令，那么可改用第二种 return 语句。divu6b 函数需要 15 条指令，而且正如代码所示，其中仅包括一条乘法指令，若用移位及加法指令实现乘法，则需要 17 条基本指令。

```
unsigned divu6a(unsigned n) {
   unsigned q, r;

   q = (n >> 3) + (n >> 5);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*6;
   return q + (11*r >> 6);
}
```

```
unsigned divu6b(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*6;
   return q + ((r + 2) >> 3);
// return q + (r > 5);
}
```

图 10.10 除数为 6 的无符号除法

对于除数更大的情况来说，通常最好是把一个和 $1/d$ 近似的数向左移动，使其小数点后的最高有效位是 1，然后用这个值来估算除法结果。这样算出来的商，其最大误差一般是 1（也许最大误差总是 1，笔者也不确定），于是就可用高效代码来实现修正步骤了。图 10.11 列出了除数为 7 和 9 时的代码，这两种情况下所使用的二进制近似值分别为：

$$4/7 \approx 0.1001\ 0010\ 0100\ 1001\ 0010\ 0100\ 1001\ 0010,$$

$$8/9 \approx 0.1110\ 0011\ 1000\ 1110\ 0011\ 1000\ 1110\ 0011$$

如果把乘以 7 和乘以 9 的操作拆开成移位与加法指令，那么这两种情况下分别需要 16 与 15 条基本指令。

```
unsigned divu7(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 4);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 2;
   r = n - q*7;
   return q + ((r + 1) >> 3);
// return q + (r > 6);
}
```

```
unsigned divu9(unsigned n) {
   unsigned q, r;

   q = n - (n >> 3);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*9;
   return q + ((r + 7) >> 4);
// return q + (r > 8);
}
```

图 10.11 除数为 7 和 9 的无符号除法

图 10.12 与 10.13 列出了除数为 10、11、12、13 的除法算法代码。它们分别基于如下二进制近似值：

8/10≈0.1100 1100 1100 1100 1100 1100 1100 1100，

8/11≈0.1011 1010 0010 1110 1000 1011 1010 0010，

8/13≈0.1010 1010 1010 1010 1010 1010 1010 1010，

8/13≈0.1001 1101 1000 1001 1101 1000 1001 1101。

```
unsigned divu10(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*10;
   return q + ((r + 6) >> 4);
// return q + (r > 9);
}
```

```
unsigned divu11(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 2) -
       (n >> 5) + (n >> 7);
   q = q + (q >> 10);
   q = q + (q >> 20);
   q = q >> 3;
   r = n - q*11;
   return q + ((r + 5) >> 4);
// return q + (r > 10);
```

图 10.12 除数为 10 和 11 的无符号除法

```
unsigned divu12(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*12;
   return q + ((r + 4) >> 4);
// return q + (r > 11);
}
```

```
unsigned divu13(unsigned n) {
   unsigned q, r;

   q = (n>>1) + (n>>4);
   q = q + (q>>4) + (q>>5);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*13;
   return q + ((r + 3) >> 4);
// return q + (r > 12);
}
```

图 10.13 除数为 12 和 13 的无符号除法

如果将乘法展开成移位和加法，那么这些函数各自需要 17、20、17、20 条基本指令。

除以 13 的情况对我们很有启发，我们由此知道了应该如何在除数倒数的二进制展开形式里寻找重复的位串。第一个赋值语句将 q 设为 n＊0.1001，第二个赋值语句为 q 加上了 n＊0.0000 1001 与 n＊0.0 0000 1001，此时 q 的值大约为 n＊0.1 0011 1011。第三条赋值语句将重复出现的位串加到了 q 之中。如早前的 divu9 函数那样，有时用减法比较好。然而，用减法时必须小心，因为估算出来的商可能过大，从而令余数为负，导致整个算法失效。想要写出最优代码颇费周折，编译器里无法集成一种可处理任意除数的通用模板算法（general cookbook method）。

上述范例所用指令都不多，一方面因为在倒数的二进制写法里，都能找出简单重复的位元组合，另一方面因为计算 r 所用的乘法，其乘数都是值比较小的常数，故而可用几条移位与加法指令实现。读者也许会问，这个办法对大一些的除数效果如何？图 10.14 与 10.15 列出了除数为 100 与 1000（这两个除数都是十进制）的除法代码，大家可以粗略评判一下。相关倒数分别为：

64/100≈0.1010 0011 1101 0111 0000 1010 0011 1101

512/1000≈0.1000 0011 0001 0010 0110 1110 1001 0111

如果将乘法展开成移位及加法，那么这两个函数分别需要 25 与 23 条基本指令。

```
unsigned divu100(unsigned n) {
   unsigned q, r;

   q = (n >> 1) + (n >> 3) + (n >> 6) - (n >> 10) +
       (n >> 12) + (n >> 13) - (n >> 16);
   q = q + (q >> 20);
   q = q >> 6;
   r = n - q*100;
   return q + ((r + 28) >> 7);
// return q + (r > 99);
}
```

图 10.14　除数为 100 的无符号除法

```
unsigned divu1000(unsigned n) {
   unsigned q, r, t;

   t = (n >> 7) + (n >> 8) + (n >> 12);
   q = (n >> 1) + t + (n >> 15) + (t >> 11) + (t >> 14);
   q = q >> 9;
   r = n - q*1000;
   return q + ((r + 24) >> 10);
// return q + (r > 999);
}
```

图 10.15　除数为 1000 的无符号除法

在除以 1000 时，倒数近似值里权重最低的 8 位几乎忽略了。图 10.15 的代码用二进制 0100 0000 取代了 1001 0111，而这样估算出的商离正确值依然不超过 1。因此，尽管看上去在大除数的倒数近似值里几乎没有重复的位元组合，但至少可以忽略某些二进制位，以便减少估算商所用的移位及加法指令。

本节以不太严密的方式向大家论证：除数为常数的无符号除法一般都可以精简为大约 20 条基本指令。要想研发一个算法，令其生成的此种代码序列（code sequence）易于集成至编译器中可没那么容易，编写最优代码的难点有三。

1. 必须在估算出的倒数里寻找重复出现的位串。

2. 有时可以用负项（像 divu10 与 divu100 那样），但用的时候必须分析其误差，而这很难做到。

3. 在估算出的倒数里，权重较低的某些位元有时可忽略（然而到底该忽略几位呢?）

对于某些目标计算机来说，还有一项困难，就是上述范例代码都有许多变体，如果计算机有很多移位及加法单元（shift and add unit），那么耗费指令较多的算法变体反而执行得更快。

图 10.7～图 10.15 所列代码均经测试，可正确处理全部 2^{32} 种被除数。

10.18.2　带符号除法

上一节所列算法可用于带符号除法。估算商时用的右移指令改为带符号右移，这相当于按“向下取整式除法”除以 2 的幂。因此，从数值上低估了商，因而余数和无符号除

法一样都是非负值。

代码算出来的结果自然会向下取整，因而需要将其修正为向 0 取整。修正方式为：若被除数为负，则向其加 $d-1$，此操作可用 3 条计算指令实现。例如除数为 6 时，将如下代码放在算法开头（其中移位操作是带符号移位）：

```
n = n + (n>>31 & 5);
```

除此之外，代码与无符号的情况非常相似，所需基本指令数通常比对应的无符号除法函数多 3 条。图 10.16 至 10.22 列出了几个范例代码，它们都经过了详尽测试。

```
int divs3(int n) {
   int q, r;

   n = n + (n>>31 & 2);            // Add 2 if n < 0.
   q = (n >> 2) + (n >> 4);        // q = n*0.0101 (approx).
   q = q + (q >> 4);               // q = n*0.01010101.
   q = q + (q >> 8);
   q = q + (q >> 16);
   r = n - q*3;                    // 0 <= r <= 14.
   return q + (11*r >> 5);         // Returning q + r/3.
// return q + (5*(r + 1) >> 4);           // Alternative 1.
// return q + ((r + 5 + (r << 2)) >> 4);// Alternative 2.
}
```

图 10.16　除数为 3 的带符号除法

```
int divs5(int n) {
   int q, r;

   n = n + (n>>31 & 4);
   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*5;
   return q + (7*r >> 5);
// return q + (r>4) + (r>9);
}
```

```
int divs6(int n) {
   int q, r;

   n = n + (n>>31 & 5);
   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 2;
   r = n - q*6;
   return q + ((r + 2) >> 3);
// return q + (r > 5);
}
```

图 10.17　除数为 5 和 6 的带符号除法

```
int divs7(int n) {
   int q, r;

   n = n + (n>>31 & 6);
   q = (n >> 1) + (n >> 4);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 2;
   r = n - q*7;
   return q + ((r + 1) >> 3);
// return q + (r > 6);
}
```

```
int divs9(int n) {
   int q, r;

   n = n + (n>>31 & 8);
   q = (n >> 1) + (n >> 2) +
      (n >> 3);
   q = q + (q >> 6);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*9;
   return q + ((r + 7) >> 4);
// return q + (r > 8);
}
```

图 10.18　除数为 7 和 9 的带符号除法

```
int divs10(int n) {
   int q, r;

   n = n + (n>>31 & 9);
   q = (n >> 1) + (n >> 2);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*10;
   return q + ((r + 6) >> 4);
// return q + (r > 9);
}
```

```
int divs11(int n) {
   int q, r;

   n = n + (n>>31 & 10);
   q = (n >> 1) + (n >> 2) -
       (n >> 5) + (n >> 7);
   q = q + (q >> 10);
   q = q + (q >> 20);
   q = q >> 3;
   r = n - q*11;
   return q + ((r + 5) >> 4);
// return q + (r > 10);
}
```

图 10.19 除数为 10 和 11 的带符号除法

```
int divs12(int n) {
   int q, r;

   n = n + (n>>31 & 11);
   q = (n >> 1) + (n >> 3);
   q = q + (q >> 4);
   q = q + (q >> 8);
   q = q + (q >> 16);
   q = q >> 3;
   r = n - q*12;
   return q + ((r + 4) >> 4);
// return q + (r > 11);
}
```

```
int divs13(int n) {
   int q, r;

   n = n + (n>>31 & 12);
   q = (n>>1) + (n>>4);
   q = q + (q>>4) + (q>>5);
   q = q + (q>>12) + (q>>24);
   q = q >> 3;
   r = n - q*13;
   return q + ((r + 3) >> 4);
// return q + (r > 12);
}
```

图 10.20 除数为 12 和 13 的带符号除法

```
int divs100(int n) {
   int q, r;

   n = n + (n>>31 & 99);
   q = (n >> 1) + (n >> 3) + (n >> 6) - (n >> 10) +
       (n >> 12) + (n >> 13) - (n >> 16);
   q = q + (q >> 20);
   q = q >> 6;
   r = n - q*100;
   return q + ((r + 28) >> 7);
// return q + (r > 99);
}
```

图 10.21 除数为 100 的带符号除法

```
int divs1000(int n) {
   int q, r, t;

   n = n + (n>>31 & 999);
   t = (n >> 7) + (n >> 8) + (n >> 12);
   q = (n >> 1) + t + (n >> 15) + (t >> 11) + (t >> 14) +
       (n >> 26) + (t >> 21);
   q = q >> 9;
   r = n - q*1000;
   return q + ((r + 24) >> 10);
// return q + (r > 999);
}
```

图 10.22 除数为 1000 的带符号除法

10.19 合计各数位求余数

本节讨论在不计算商的前提下，如何求除法余数。本节所述算法仅适用于 $2^k \pm 1$ 形式的除数，其中 k 是大于等于 2 的整数。大部分情况下，算法都要先执行一个相当短的计算过程，然后查表（用“索引式载入指令”（indexed load instruction）实现）。

后文将反复用到下面这条同余的基本性质：

定理 C　若 $a \equiv b \pmod m$ 且 $c \equiv d \pmod m$，则：

$$a+c \equiv b+d \pmod m，且$$
$$ac \equiv bd \pmod m。$$

无符号的情况比较简单，所以先讲。

10.19.1 求无符号除法的余数

对于除数为 3 的情况，可以给 $1 \equiv 1 \pmod 3$ 这个简单的同余式反复乘上 $2 \equiv -1 \pmod 3$，于是由定理 C 可得：

$$2^k = \begin{cases} 1 \pmod 3，k 为偶数 \\ -1 \pmod 3，k 为奇数 \end{cases}$$

因此，将 n 写为二进制形式 $\ldots b_3\, b_2\, b_1\, b_0$ 之后，反复运用定理 C，即可得出下式：

$$n = \ldots + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0 \equiv \ldots - b_3 + b_2 - b_1 + b_0 \pmod 3$$

由此可知，交替加减二进制形式中的各个位元，即可获得一个较小的数，而此数除以 3 的余数和 n 除以 3 的余数相同。如果合计各个数位之后得到负值，那么必须再加上 3 的倍数，以使其非负。可反复执行此过程，直至结果位于 0 至 2 之间。

也可用此技巧求十进制数除以 11 的余数。

因此，如果计算机有种群计数指令，那么在编写函数计算无符号数 n 除以 3 的余数时，可用下列代码开头：

```
n = pop(n & 0x55555555) - pop(n & 0xAAAAAAAA);
```

Paolo Bonzini 先生发现了下面这个奇妙的等式 [Bonz]，利用此式可化简上述代码：

$$\text{pop}(\boldsymbol{x}\ \&\ \bar{\boldsymbol{m}}) - \text{pop}(\boldsymbol{x}\ \&\ \boldsymbol{m}) = \text{pop}(\boldsymbol{x} \oplus \boldsymbol{m}) - \text{pop}(\boldsymbol{m}) \qquad (2)$$

证明如下：

$\text{pop}(\boldsymbol{x}\ \&\ \bar{\boldsymbol{m}}) - \text{pop}(\boldsymbol{x}\ \&\ \boldsymbol{m})$	
$= \text{pop}(\boldsymbol{x}\ \&\ \bar{\boldsymbol{m}}) - (32 - \text{pop}(\overline{\boldsymbol{x}\ \&\ \boldsymbol{m}}))$	$\text{pop}(\boldsymbol{a}) = 32 - \text{pop}(\bar{\boldsymbol{a}})$
$= \text{pop}(\boldsymbol{x}\ \&\ \bar{\boldsymbol{m}}) + \text{pop}(\bar{\boldsymbol{x}} \mid \bar{\boldsymbol{m}}) - 32$	根据德摩根定律
$= \text{pop}(\boldsymbol{x}\ \&\ \bar{\boldsymbol{m}}) + \text{pop}(\bar{\boldsymbol{x}}\ \&\ \boldsymbol{m}) + \text{pop}(\bar{\boldsymbol{m}}) - 32$	$\text{pop}(\boldsymbol{a} \mid \boldsymbol{b}) = \text{pop}(\boldsymbol{a}\ \&\ \bar{\boldsymbol{b}}) + \text{pop}(\boldsymbol{b})$

$=\text{pop}((x\ \&\bar{m})|(\bar{x}\ \&m))+\text{pop}(\bar{m})-32$ 　　因为 $x\ \&\bar{m}$ 与 $\bar{x}\ \&m$ 里面的“1”不重合，所以用“或”运算取代加法

$=\text{pop}(x\oplus m)-\text{pop}(m)$

由于推导过程中的 32（也就是字长）已经消去，所以此式对任意字长均成立。还有一种办法也能证明（2）式。首先观察到此式在 $x=\mathbf{0}$ 时成立。然后假设要把 x 中某个位置上的位元由 0 变为 1，此时若 m 对应位置上是 1，则（2）式左右两边都比原来少 1，等式依然成立；若 m 对应位置上是 0，则（2）式左右两边都比原来多 1，等式依然成立。

对刚才那行 C 语言代码运用（2）式可得：

```
n = pop(n ^ 0xAAAAAAAA) - 16;
```

反复执行此变换，尽可能把 n 调整到 0 和 2 之间。最好不要算出负的 n 值，因为下一轮无法正确处理符号位。可以给 n 加上 3 的倍数，只要此加数足够大，就能避免负值。图 10.23 列出了 Bonzini 所写的代码，其中给运算过的 n 加上了常数 39。此数比刚好能令 n 非负的那个值还要大，不过它还可以使第二轮算出来的 n 位于 −3 与 2 之间（而非 −3 与 3 之间）[⊖]。这就简化了 return 语句：只需在 n 为负数时加 3 即可。算上加载大常数所用的两条指令，此函数共执行 11 条指令。

```
int remu3(unsigned n) {
   n = pop(n ^ 0xAAAAAAAA) + 23;     // Now 23 <= n <= 55.
   n = pop(n ^ 0x2A) - 3;            // Now -3 <= n <= 2.
   return n + (((int)n >> 31) & 3); // (signed shift)
}
```

图 10.23　用种群计数指令求无符号数除以 3 的余数

图 10.24 列出了一种变体算法，只需 4 条指令，外加一次简单的查表操作（比方说，可以用带索引的字节载入指令（indexed load byte instrunction）实现）。

```
int remu3(unsigned n) {

   static char table[33] = {2, 0,1,2, 0,1,2, 0,1,2,
          0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
          0,1,2, 0,1};

   n = pop(n ^ 0xAAAAAAAA);
   return table[n];
}
```

图 10.24　用种群计数及查表法求无符号数除以 3 的余数

还有一种不使用种群计数指令的办法。因为 $4\equiv1\pmod 3$，所以 $4^k\equiv1\pmod 3$。将二进制数中每两个位元编为一对，把 00 与 11 之间的 4 种数值分别写为四进制数中的 0

[⊖] 刚好令 n 非负的那个常数是 18，如果使用它，那么第一轮算出的 n 在 2 与 34 之间，而第二轮和 0x2A（二进制 101010）取异或时，如果 n 是 21（二进制 010101），那么异或出来的数是二进制 111111，种群计数为 6，减去 3 后是 3，不在 −3 与 2 之间，无法简化 return 语句。——译者注

至 3，这样整个数字就变成四进制数了。于是，可以参考 5.1 节的图 5.2，省去其第一行可执行代码，然后统计数位和（加法操作不会溢出）。最终合计结果在 0 至 48 之间，用查表法可将其调整至 0 与 2 之间。这样写出来的函数需要 16 条基本指令，外加一条索引式载入（indexed load）指令。

还有一种类似的办法，其效果稍微好些。首先，可以把 n 化成另外一个较小的数，使这两个数除以 3 的余数相同：

```
n = (n >> 16) + (n & 0xFFFF);
```

这行代码把 n 分成了左右两个 16 位部分，然后将其相加。n 的左 16 位在不移动之前的数值与移动之后的数值相比，除以 3 所得的余数未变，因为移动之后的数再乘以 2^{16} 就等于移动之前的数值，而 $2^{16}\equiv 1 \pmod 3$，所以根据定理 C 可知移位前后的两个数关于模 3 同余。更一般的规律是：若 k 为偶数，则 $2^k\equiv 1 \pmod 3$。图 10.25 的代码将这一规律套用了 5 次，代码共需 19 条指令。如果像图 10.26 这样，提前终止数位加总操作，并使用内存查表法（in-memory table lookup），那么还可减少指令数（9 条指令，外加 1 条索引式载入指令）。若使用大小为 0x2FE=766 字节的表格，则可将指令数减少为 6 条（外加 1 条索引式载入指令）。

图 10.27 的代码使用 $16^k\equiv 1 \pmod 5$ 及 $4\equiv -1 \pmod 5$ 这两个关系式来计算无符号数除以 5 的余数。如果用移位及加法来实现乘以 3，则需要 21 条基本指令。

```
int remu3(unsigned n) {
   n = (n >> 16) + (n & 0xFFFF);        // Max 0x1FFFE.
   n = (n >>  8) + (n & 0x00FF);        // Max 0x2FD.
   n = (n >>  4) + (n & 0x000F);        // Max 0x3D.
   n = (n >>  2) + (n & 0x0003);        // Max 0x11.
   n = (n >>  2) + (n & 0x0003);        // Max 0x6.
   return (0x0924 >> (n << 1)) & 3;
}
```

图 10.25　通过合计数位及寄存器查表法求无符号数除以 3 的余数

```
int remu3(unsigned n) {
   static char table[62] = {0,1,2, 0,1,2, 0,1,2, 0,1,2,
        0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
        0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
        0,1,2, 0,1,2, 0,1};

   n = (n >> 16) + (n & 0xFFFF);        // Max 0x1FFFE.
   n = (n >>  8) + (n & 0x00FF);        // Max 0x2FD.
   n = (n >>  4) + (n & 0x000F);        // Max 0x3D.
   return table[n];
}
```

图 10.26　通过合计数位及内存查表法求无符号数除以 3 的余数

使用与图 10.26 类似的查表法，可以节省指令。实际上除以 5 的代码和除以 3 一样，只是表格不同：

```
static char table[62] = {0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1};
```

```
int remu5(unsigned n) {
   n = (n >> 16) + (n & 0xFFFF);            // Max 0x1FFFE.
   n = (n >>  8) + (n & 0x00FF);            // Max 0x2FD.
   n = (n >>  4) + (n & 0x000F);            // Max 0x3D.
   n = (n>>4) - ((n>>2) & 3) + (n & 3);     // -3 to 6.
   return (01043210432 >> 3*(n + 3)) & 7;  // Octal const.
}
```

图 10.27　通过合计数位法求无符号数除以 5 的余数

图 10.28 的代码使用关系式 $8^k \equiv 1 \pmod 7$ 来计算无符号数除以 7 的余数（需要 9 条指令，外加 1 条索引式载入指令）。

```
int remu7(unsigned n) {

   static char table[75] = {0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
             0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
             0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
             0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4};

   n = (n >> 15) + (n & 0x7FFF);         // Max 0x27FFE.
   n = (n >>  9) + (n & 0x001FF);        // Max 0x33D.
   n = (n >>  6) + (n & 0x0003F);        // Max 0x4A.
   return table[n];
}
```

图 10.28　通过合计数位法求无符号数除以 7 的余数

最后再举个例子，图 10.29 的代码可以计算无符号数除以 9 的余数。此代码基于 $8 \equiv -1 \pmod 9$ 这一关系式。它需要 9 条基本指令，外加 1 条索引式载入指令。用长度为 831（十进制）的表格可将基本指令数减少至 6 条。

```
int remu9(unsigned n) {

   int r;
   static char table[75] = {0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
         0,1,2,3,4,5,6,7,8, 0,1,2};

   r = (n & 0x7FFF) - (n >> 15);       // FFFE0001 to 7FFF.
   r = (r & 0x01FF) - (r >>  9);       // FFFFFFC1 to 2FF.
   r = (r & 0x003F) + (r >>  6);       // 0 to 4A.
   return table[r];
}
```

图 10.29　通过合计数位法求无符号数除以 9 的余数

10.19.2　求带符号除法的余数

可改编前一节所用的合计数位法（digit summing method），以计算带符号除法的余数。与无符号除法类似，也需要用几个步骤来修正运算结果，此外似乎没有更好的办法了。必须执行以下两项修正：（1）由于对符号位解读不同而产生的误差必须修正，

(2) 为运算结果加上或减去除数 d 的倍数，以便将其调整至 0 与 $-(d-1)$ 之间。

在除数为 3 时，如果被除数 n 的符号位是 1，那么求无符号除法余数的代码就会认为，这一位对余数的影响为 2（因为 $2^{31} \bmod 3=2$）。对于带符号除法的余数来说，符号位对其影响值仅为 1（因为 $(-2^{31}) \bmod 3=1$）。因此，可以先套用无符号的余数算法，然后再给结果减 1，以修正其值。最后，必须把结果调整到 0 与 -2 之间。也就是说，执行完无符号的余数算法代码后，必须按下列方式调整其结果：

$$(0,1,2) \Rightarrow (-1,0,1) \Rightarrow (-1,0,-2)$$

调整余数有种简便方法：如果无符号算法求出的余数是 0 或 1，就减 1，如果是 2，就减 4（在被除数为负时）。代码不能改动被除数 n，因为最后一步要用到此值。

该过程可套用在所有无符号版本的对 3 求余函数上。例如，改编 10.19.1 节中的图 10.26，可得图 10.30。此算法共需 13 条基本指令，外加 1 条索引式载入指令。继续增大表格，可减少指令数。

```
int rems3(int n) {
   unsigned r;
   static char table[62] = {0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
       0,1,2, 0,1,2, 0,1};

   r = n;
   r = (r >> 16) + (r & 0xFFFF);          // Max 0x1FFFE.
   r = (r >>  8) + (r & 0x00FF);          // Max 0x2FD.
   r = (r >>  4) + (r & 0x000F);          // Max 0x3D.
   r = table[r];
   return r - (((unsigned)n >> 31) << (r & 2));
}
```

图 10.30　用合计数位法求带符号数除以 3 的余数

与之相似，图 10.31 至图 10.33 分别列出了除数为 5、7、9 的带符号数求余代码。这三个函数都需要 15 条基本指令，外加 1 条索引式载入指令。它们都使用带符号右移操作，在最后的修正步骤中，如果被除数为负而余数非 0，则要从余数中减掉除数。若表格变大，则所需指令数会减少。

```
int rems5(int n) {
   int r;
   static char table[62] = {2,3,4, 0,1,2,3,4, 0,1,2,3,4,
             0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
             0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
             0,1,2,3,4, 0,1,2,3};

   r = (n >> 16) + (n & 0xFFFF);     // FFFF8000 to 17FFE.
   r = (r >>  8) + (r & 0x00FF);     // FFFFFF80 to 27D.
   r = (r >>  4) + (r & 0x000F);     // -8 to 53 (decimal).
   r = table[r + 8];
   return r - (((int)(n & -r) >> 31) & 5);
}
```

图 10.31　用合计数位法求带符号数除以 5 的余数

```
int rems7(int n) {
   int r;
   static char table[75] =    {5,6, 0,1,2,3,4,5,6,
     0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
     0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
     0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2};

   r = (n >> 15) + (n & 0x7FFF);    // FFFF0000 to 17FFE.
   r = (r >>  9) + (r & 0x001FF);   // FFFFFF80 to 2BD.
   r = (r >>  6) + (r & 0x0003F);   // -2 to 72 (decimal).
   r = table[r + 2];
   return r - (((int)(n & -r) >> 31) & 7);
}
```

图 10.32　用合计数位法求带符号数除以 7 的余数

```
int rems9(int n) {
   int r;
   static char table[75] = {7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0};

   r = (n & 0x7FFF) - (n >> 15);    // FFFF7001 to 17FFF.
   r = (r & 0x01FF) - (r >>  9);    // FFFFFF41 to 0x27F.
   r = (r & 0x003F) + (r >>  6);    // -2 to 72 (decimal).
   r = table[r + 2];
   return r - (((int)(n & -r) >> 31) & 9);
}
```

图 10.33　用合计数位法求带符号数除以 9 的余数

10.20　用乘法及右移位求余数

本节所述算法原则上适用于比 2 大的所有整数除数，不过从实际角度讲，只有当除数较小或具备 2^k-1 的格式时，效果才好。与上一节相同，大部分代码都先执行较短的计算过程，然后查表。

10.20.1　求无符号除法的余数

本节所说的 $a \bmod b$ 是数学记法（而非计算机代数记法），它表示符合 $x \equiv a \pmod b$ 的整数 x，其中 a、b 为整数且 $b>0$，而 $0 \leqslant x < b$。

要计算 $n \bmod 3$ 的值，首先来看下列不等式：

$$n \bmod 3 = \left\lfloor \frac{4}{3}n \right\rfloor \bmod 4 \tag{3}$$

证明：设 $n=3k+\delta$，其中 δ 与 k 都是整数，且 $0 \leqslant \delta \leqslant 2$。于是可得

$$\left\lfloor \frac{4}{3}(3k+\delta) \right\rfloor \bmod 4 = \left\lfloor 4k+\frac{4\delta}{3} \right\rfloor \bmod 4 = \left\lfloor \frac{4\delta}{3} \right\rfloor \bmod 4$$

显然，当 δ 等于 0，1，2 时，上面最后一个表达式也分别等于 0，1，2。于是，可以把对 3 求余问题变成对 4 求余问题，这对使用二进制数的计算机来说，无疑简单多了。 ■

关系式（3）未必对所有“模数”都成立，然而只要“模数”具备形式 2^k-1（k 为大于 1 的整数），那么就有类似的关系式成立。例如，很容易证明

$$n \bmod 7 = \left\lfloor \frac{8}{7} n \right\rfloor \bmod 8$$

若除数不是 2^k-1，则没有与之类似的简单关系式，不过，有个独特的性质可用来计算除以这些除数后的余数。比方说，如果除数是十进制的 10，那么考虑下式：

$$\left\lfloor \frac{16}{10} n \right\rfloor \bmod 16 \tag{4}$$

设 $n=10k+\delta$，其中 $0\leqslant\delta\leqslant 9$。那么

$$\left\lfloor \frac{16}{10} n \right\rfloor \bmod 16 = \left\lfloor \frac{16}{10}(10k+\delta) \right\rfloor \bmod 16 = \left\lfloor \frac{16\delta}{10} \right\rfloor \bmod 16$$

当 $\delta=0$，1，2，3，4，5，6，7，8，9 时，上面最末一个表达式的值分别为 0，1，3，4，6，8，9，11，12，14，这些数值各不相同。因此，如果有种简单办法能算出（4）式，那么就可以将 0 对译为 0，将 1 对译为 1，将 3 对译为 2，将 4 对译为 3，……以此推算 n 除以 10 的余数。不过这需要用到“转换表”（translation table），其长度是大于等于除数的下一个 2 的幂，所以只有在除数相当小的时候，此办法才实用（如果除数是 2^k-1，则不需查表）。

下面要演示的代码对上述理论稍加运用，并多次“试错”。

考虑无符号数对 3 求余的情况。按照（3）式，需要知道 $4n/3$ 整数部分最右侧两个位元。先乘以 $\lfloor 2^{32}/3 \rfloor$，再通过右移指令除以 2^{30}，即可估算出此值。乘以 $\lfloor 2^{32}/3 \rfloor$ 之后（假设使用只能求出乘积低 32 位的那种乘法指令），会丢弃高权重位元。不过那些位元无关紧要，而且丢弃了也好，因为我们只想计算对 4 求余的值。因为 $\lfloor 2^{32}/3 \rfloor = \text{0x55555555}$，所以也许能按下式计算：

$$\boldsymbol{r} \leftarrow (\mathbf{0x5555\ 5555} * \boldsymbol{n}) \overset{u}{\gg} 30$$

经实验，此办法对 0 至 $2^{30}+2$ 之间的 $\boldsymbol{n}$ 成立。应该说，这个办法基本能用。如果 $\boldsymbol{n}$ 非 0 而且是 3 的倍数，那么算出的结果是 3。因此，算完之后要再加一步，把（0，1，2，3）对译为（0，1，2，0）。

为了扩大此办法的适用范围，乘法必须更精确。再精确两个位元就够了（也就是说，和 **0x5555 5555.4** 相乘）。先用下式计算，然后再转换其结果，这样就能应对任意 32 位无符号整数 $\boldsymbol{n}$ 了。

$$\boldsymbol{r} \leftarrow (\mathbf{0x5555\ 5555} * \boldsymbol{n} + (\boldsymbol{n} \overset{u}{\gg} 2)) \overset{u}{\gg} 30$$

当然，有办法能严格证明此式，然而其算术过程太冗长，而且易出错。

在大部分计算机上，转换步骤需要 3 至 4 条指令，不过有种办法能够省去此步骤，但要多花两条指令。上述表达式算出来的 $\boldsymbol{r}$ 偏低。如果稍微往高算一点，那么结果就总是 0，1，2 了。于是可写出图 10.34 中的 C 语言函数（需要 8 条指令，其中 1 条是乘法）。

```
int remu3(unsigned n) {
   return (0x55555555*n + (n >> 1) - (n >> 3)) >> 30;
}
```

图 10.34 用乘法计算无符号数除以 3 的余数

如果把乘法展开成移位和加法指令，那么就有了图 10.35 中的这个函数，它需要 13 条指令。

```
int remu3(unsigned n) {
   unsigned r;

   r = n + (n << 2);
   r = r + (r << 4);
   r = r + (r << 8);
   r = r + (r << 16);
   r = r + (n >> 1);
   r = r - (n >> 3);
   return r >> 30;
}
```

图 10.35 用展开的乘法计算无符号数除以 3 的余数

无符号数对 5 求余的办法与对 3 求余很相似。设 $n=5k+r$，其中 $0\leqslant r\leqslant 4$，那么 $(8/5)n \bmod 8=(8/5)(5k+r) \bmod 8=(8/5)r \bmod 8$。当 $r=0$，1，2，3，4 时，$(8/5)r \bmod 8$ 的值分别为 0，1，3，4，6。由于 $\lfloor 2^{32}/5 \rfloor$=0x3333 3333，所以可写出图 10.36 这样的函数（11 条指令，其中 1 条是乘法）。最后一步（也就是返回语句）分别将（0，1，3，4，6，7）映射为（0，1，2，3，4，0），代码用的是寄存器内（in-register）查表法，而不是用索引式载入指令从内存中查表。如果把 2 映射为 2，将 5 映射为 4，那么在乘以 $2^{32}/5$ 时，由乘数的小数部分（十六进制为 0.333…）所引发的误差就可以用n≫3来估算了。省掉 n≫3 这个“精确项”（“accuracy” term）之后，代码仍然适用于 0 至 0x6000 0004 之间的 n。

```
int remu5(unsigned n) {
   n = (0x33333333*n + (n >> 3)) >> 29;
   return (0x04432210 >> (n << 2)) & 7;
}
```

图 10.36 用乘法计算无符号数除以 5 的余数

计算无符号数对 7 求余所用的代码与之相似，而且映射步骤更简单，只需把 7 变成 0 就好。图 10.37 列出了一种编码方式（需要 11 条指令，其中 1 条为乘法）。省掉“精确项”n≫4 之后，代码依然适用于 0 至 0x4000 0006 之间的 n 值。如果把两个精确项都省去，那么代码能处理的最大 n 值是 0x0800 0006。

```
int remu7(unsigned n) {
   n = (0x24924924*n + (n >> 1) + (n >> 4)) >> 29;
   return n & ((int)(n - 7) >> 31);
}
```

图 10.37 用乘法计算无符号数除以 7 的余数

图 10.38 列出了计算无符号数对 9 求余所用的代码。它需要 6 条指令，其中 1 条是乘法，除了这 6 条之外，还需 1 条索引式载入指令。如果省去 n≫1 这项，并将乘数改为 0x1C71 C71D，那么函数能处理的最大 n 值是 0x1999 999E。

图 10.39 列出了计算无符号数对 10 求余的一种办法。它需要 8 条指令，其中 1 条是乘法，除了这 8 条之外，还需 1 条索引式载入指令。若将 n≫3 这项省去，则至多能应对 n 为 0x4000 0004 的情况。如果两个精确项都省去，那么能够处理的最大 n 值是 0x0AAA AAAD。

最后再举个例子，思考一下如何对 63 求余。5.1 节的第二个 pop 函数在计算种群计数时会用到此函数。Joe Keane 先生编出了图 10.40 这段相当神奇的代码［Keane］。在基

本 RISC 架构中，它需要 12 条基本指令。

```
int remu9(unsigned n) {
   static char table[16] = {0, 1, 1, 2, 2, 3, 3, 4,
                            5, 5, 6, 6, 7, 7, 8, 8};

   n = (0x1C71C71C*n + (n >> 1)) >> 28;
   return table[n];
}
```

图 10.38　用乘法计算无符号数除以 9 的余数

```
int remu10(unsigned n) {
   static char table[16] = {0, 1, 2, 2, 3, 3, 4, 5,
                            5, 6, 7, 7, 8, 8, 9, 0};

   n = (0x19999999*n + (n >> 1) + (n >> 3)) >> 28;
   return table[n];
}
```

图 10.39　用乘法计算无符号数除以 10 的余数

```
int remu63(unsigned n) {
   unsigned t;

   t = (((n >> 12) + n) >> 10) + (n << 2);
   t = ((t >> 6) + t + 3) & 0xFF;
   return (t - (t >> 6)) >> 2;
}
```

图 10.40　用 Keane 算法求无符号数除以 63 的余数

```
int remu63(unsigned n) {
   n = (0x04104104*n + (n >> 4) + (n >> 10)) >> 26;
   return n & ((n - 63) >> 6);    // Change 63 to 0.
}
```

图 10.41　用乘法求无符号数除以 63 的余数

图 10.41 的代码用“相乘并右移”的办法。它需要 11 条基本 RISC 指令，其中 1 条是乘法。除非计算机能极快算出乘法，并把加载常数 0x0410 4104 这一操作搬到循环外面，否则此办法不如 Keane 的快。

在某些计算机上可以改进此算法，把乘法展开成移位与加法（整个函数需要 15 条基本指令）：

```
r = (n << 2) + (n << 8);        // r = 0x104*n.
r = r + (r << 12);              // r = 0x104104*n.
r = r + (n << 26);              // r = 0x04104104*n.
```

10.20.2　求带符号除法的余数

与“合计数位法”（digit summing method）类似，也可改编“相乘右移法”，用以计算带符号除法的余数。除了在无符号算法后面加几个步骤来修正计算结果之外，似乎也没有更好的办法了。比方说，可以把 10.20.1 节的图 10.34 改编为图 10.42（需要 12 条指令，其中 1 条是乘法）。

图 10.43 至图 10.46 列出了带符号数对 5、7、9、10 求余的可行算法。除数为 7 的算

法多用了好几条指令（共计19条，其中1条是乘法），也许像除数为5、9、10的算法那样改用查表法会更好些。若使用查表法，则所用表格会比无符号除法时大一倍，计算查表索引时，要把除数的符号位考虑进来。值为u的元素未使用。

```
int rems3(int n) {
   unsigned r;

   r = n;
   r = (0x55555555*r + (r >> 1) - (r >> 3)) >> 30;
   return r - (((unsigned)n >> 31) << (r & 2));
}
```

图10.42 用乘法计算带符号数除以3的余数

```
int rems5(int n) {
   unsigned r;
   static signed char table[16] = {0, 1, 2, 2, 3, u, 4, 0,
                                   u, 0,-4, u,-3,-2,-2,-1};

   r = n;
   r = ((0x33333333*r) + (r >> 3)) >> 29;
   return table[r + (((unsigned)n >> 31) << 3)];
}
```

图10.43 用乘法计算带符号数除以5的余数

```
int rems7(int n) {
   unsigned r;

   r = n - (((unsigned)n >> 31) << 2);  // Fix for sign.
   r = ((0x24924924*r) + (r >> 1) + (r >> 4)) >> 29;
   r = r & ((int)(r - 7) >> 31);        // Change 7 to 0.
   return r - (((int)(n&-r) >> 31) & 7);// Fix n<0 case.
}
```

图10.44 用乘法计算带符号数除以7的余数

```
int rems9(int n) {
   unsigned r;
   static signed char table[32] = {0, 1, 1, 2, u, 3, u, 4,
                                   5, 5, 6, 6, 7, u, 8, u,
                                  -4, u,-3, u,-2,-1,-1, 0,
                                   u,-8, u,-7,-6,-6,-5,-5};

   r = n;
   r = (0x1C71C71C*r + (r >> 1)) >> 28;
   return table[r + (((unsigned)n >> 31) << 4)];
}
```

图10.45 用乘法计算带符号数除以9的余数

```
int rems10(int n) {
   unsigned r;
   static signed char table[32] = {0, 1, u, 2, 3, u, 4, 5,
                                   5, 6, u, 7, 8, u, 9, u,
                                  -6,-5, u,-4,-3,-3,-2, u,
                                  -1, 0, u,-9, u,-8,-7, u};
   r = n;
   r = (0x19999999*r + (r >> 1) + (r >> 3)) >> 28;
   return table[r + (((unsigned)n >> 31) << 4)];
}
```

图10.46 用乘法计算带符号数除以10的余数

10.21 将普通除法化为精确除法

既然在不计算商的情况下就能求得余数，那么还有一种方法可以算出 $\lfloor n/d \rfloor$ 的商 q：先算出余数，再将其从被除数 n 中减去，然后把二者之差与除数 d 相除。因为最后执行的是“精确除法”（exact division，整除），所以可用与除数 d 的“乘法逆元素”相乘来代替（参见 10.16 节）。在既要计算商又要计算余数时，这个方法尤其好用。

现在试着用此思路来计算无符号数与 3 相除。以相乘的办法算出余数，然后再求商，于是就有了图 10.47 中的这份代码。

```
unsigned divu3(unsigned n) {
   unsigned r;

   r = (0x55555555*n + (n >> 1) - (n >> 3)) >> 30;
   return (n - r)*0xAAAAAAAB;
}
```

图 10.47 通过“精确除法”计算无符号数除以 3 的余数及商

此算法需要 11 条指令，其中有两条是和大数相乘。（将常数 0xAAAA AAAB 右移一位，即可得到常数 0x5555 5555。）与之相比，更为直观的算法需要 14 条指令才能算出商数 q（例如 10.18.1 节的图 10.8），其中两条是与较小的常数相乘，若将乘法展开成移位与加法，则需 17 条基本指令。如果还需算出余数，那么可以按 r = n - q * 3 来算，于是，刚才那种更为直观的算法就需要 16 条指令了，其中 3 条是与较小的常数相乘，若用移位与加法来取代乘法，则需 20 条基本指令。

如果把图 10.47 中的乘法展开成移位和加法指令，那么这个算法就不太吸引人了，因为现在要 24 条基本指令才行。由此可见，要是一台计算机没有“multiply high”指令，但却能快速执行“模 2^{32}”乘法指令，同时除法操作又很慢，那么这种“精确除法”算法也许效果不错，若还能轻松应对大常数，则效果尤佳。

对于除数为 3 的带符号除法来说，“精确除法”算法的代码如图 10.48 所示。它需要 15 条指令，其中两条是与大常数相乘。

```
int divs3(int n) {
   unsigned r;

   r = n;
   r = (0x55555555*r + (r >> 1) - (r >> 3)) >> 30;
   r = r - (((unsigned)n >> 31) << (r & 2));
   return (n - r)*0xAAAAAAAB;
}
```

图 10.48 使用“精确除法”计算带符号数除以 3 的余数及商

最后来看图 10.49，这段代码可算出无符号数除以 10 的商和余数。它需要 12 条指令，其中两条是和大常数相乘，外加一条索引式载入指令。

```
unsigned divu10(unsigned n) {
   unsigned r;
   static char table[16] = {0, 1, 2, 2, 3, 3, 4, 5,
                            5, 6, 7, 7, 8, 8, 9, 0};

   r = (0x19999999*n + (n >> 1) + (n >> 3)) >> 28;
   r = table[r];
   return ((n - r) >> 1)*0xCCCCCCCD;
}
```

图 10.49 使用“精确除法”计算无符号数除以 10 的余数及商

10.22 计时测试

很多计算机都有 32×32⇒64 乘法指令，所以当除数为 3 这样的常数时，读者也许觉得 10.8.1 节的算法最快。要是没有这样的乘法指令，但计算机却有另外一种 32×32⇒32 的快速乘法指令，那么“精确除法”算法可能比较好，它适用于除法慢而乘法快的计算机。为验证此猜测，笔者写了一段汇编语言程序，用以比较除数为 3 的 4 种算法。测试结果列在表 10.4 中。测试所用计算机的 CPU 是 667 兆赫的奔腾 3（Pentium Ⅲ）（2000 年左右上市），在其他一些计算机中的测试结果应该与之类似。

表 10.4 在奔腾 3 计算机中测试无符号数除以 3 的各种算法

除法算法	周期
使用计算机除法指令（divl）	41.08
用 32×32⇒64 乘法指令（10.8.1 节的算法）	4.28
全部使用基本指令（10.18.1 节的图 10.8）	14.10
将普通除法化为“精确除法”（10.21 节的图 10.47）	6.68

第一行统计了两条指令的执行周期：清空 64 位源寄存器左半边所用的 xorl 指令和计算除法所用的 divl 指令，两者合计显然需要四十多个周期。第二行也统计了两条指令的耗时：乘法及右移 1 位指令（mull 及 shrl）。第三行统计了 21 条基本指令的总耗时。它使用 10.18.1 节的图 10.8，并用第二种可选方案替换了其中的返回语句，同时用一条指令（leal[⊖]）计算与 3 的乘积。由于计算机主要使用“两地址格式”（two-address）的指令，所以还得使用若干条 move 指令。最后一行统计了 10 条指令的总耗时，其中两条是乘法（imull），其余都是基本指令。这两条 imull 指令都使用 4 字节的常量字段（immediate field）来编码大常数。（此处使用了带符号乘法指令 imull 而未使用无符号版的 mull，因为前者算出的低权重 32 位与后者相同，而且寻址模式（addressing mode）更多。）

如果要同时计算商和余数，那么“精确除法”算法甚至比第 2 个和第 3 个算法都好，因为若是用那两种方法计算，还需再用几条指令算出 $r \leftarrow n - q * 3$ 才行。（divl 指令在计

⊖ 此指令本来是“载入有效地址”（Load Effective Address）所用，不过也常用来实现加法与乘法。详情可参考：http://stackoverflow.com/questions/13329124。——译者注

算商的同时也能求出余数。)

10.23　用电路计算除数为 3 的除法

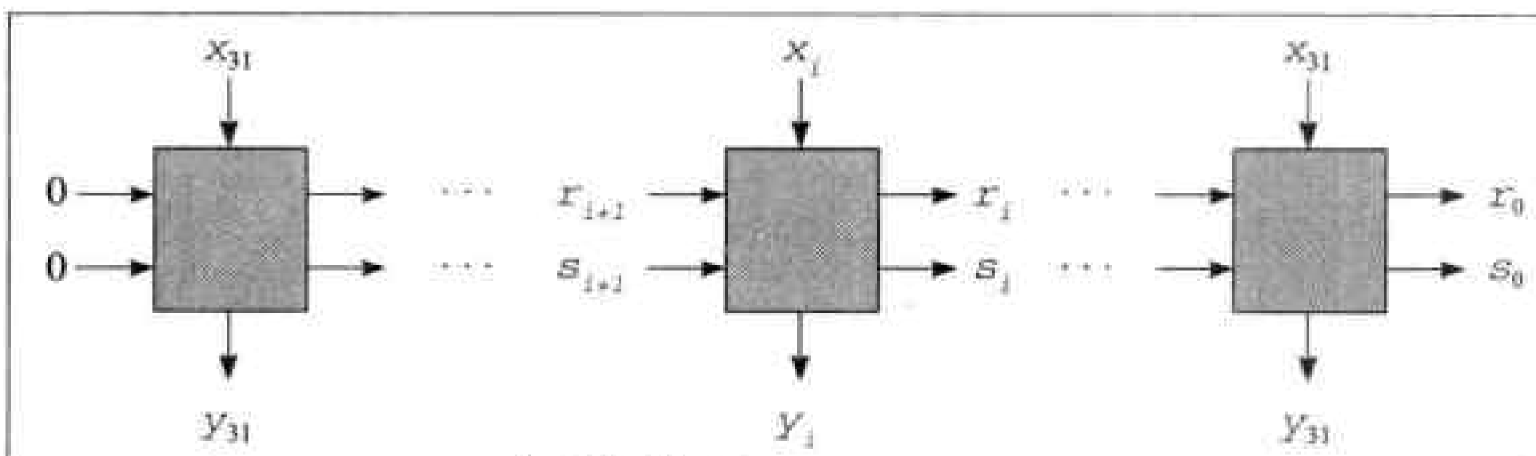
有种和加法器同样简单的电路能计算除数为 3 的除法。构建此电路的方法与通过 n 个 1 位“全加器”(full adder) 电路来构建 n 位加法器的基本方法非常相似，只不过除法器的信号是从最高有效位流向最低有效位。

思考一下如何用小学生竖式法来除以 3，只不过这次用二进制而不用十进制。把上个阶段的余数 0，1 或 2 放在当前这一位的左边，拿这个联合之后的数字去除以 3，于是就能算出商中的某一位了。表 10.5 列出了此算法的逻辑。余数用 r_i 和 s_i 这两个位元表示，r_i 是最高有效位。余数不可能是 3，所以表格最后两行无关紧要（“don't care” cases)。

表 10.5　除以 3 的逻辑

r_{i+1}	s_{i+1}	x_i	y_i	r_i	s_i
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	—	—	—
1	1	1	—	—	—

图 10.50 演示了除数为 3 的 32 位除法电路。表示商数的字组由 y_{31} 至 y_0 这 32 个位元组成，余数是 $2r_0+s_0$。



$$y_i = r_{i+1} + s_{i+1}x_i$$

$$r_i = \overline{r_{i+1}}s_{i+1}\overline{x_i} + r_{i+1}x_i$$

$$s_i = \overline{r_{i+1}}\,\overline{s_{i+1}}x_i + r_{i+1}\overline{s_{i+1}}\,\overline{x_i}$$

图 10.50　用逻辑电路实现除数为 3 的除法

想用硬件实现“除以 3 操作”(divide-by-3 operation) 还有一种办法：通过适当的舍入及缩放操作，设法将被除数和 3 的倒数（二进制 0.0101 01...）相乘。这也是 10.3.1 和 10.8.1 节用的技巧。

10.24 习题

1. 请证明，对于除数为偶数的无符号除法来说，如果提前执行下列任意一项操作，那么就无需再使用 shrxi 指令（或其等效代码）了：(a) 用“与”操作把被除数的最低有效位清空 [CarWer]，(b) 用右移 1 位指令把被除数除以 2，然后再除以原除数的一半。
2. 编写一个与 10.15 节图 10.4 相似的函数，计算带符号除法所用的神奇数字。只考虑除数为正的情况。
3. 请演示如何用“牛顿法”算出整数 d 对于模 81 的“乘法逆元素”。计算 $d=146$ 时的结果。

I think that I shall never envision	天生除法，性情乖张，
An op unlovely as division.	如欲安宁，避其锋芒。
An op whose answer must be guessed	左加右乘，始得有商，
And then, through multiply, assessed;	寻彼正果，费尽思量。
An op for which we dearly pay,	流年似水，百事皆荒，
In cycles wasted every day.	昼思夜想，整日彷徨。
Division code is often hairy;	精雕细琢，筹算未央，
Long division's downright scary.	千头万绪，无分短长。
The proofs can overtax your brain,	皓首穷经，白发苍苍，
The ceiling and floor may drive you insane.	上下求索，为伊痴狂。
Good code to divide takes a Knuthian hero,	壮哉高氏，代码铿锵，
But even God can't divide by zero![⊖]	敢问神明，除零何妨？

⊖ 原诗中文大意为：我从未见过除法这般不讨喜的操作符。答案必须要猜，求商基本要乘。这个操作代价高，每天要花大量 CPU 周期。计算普通除法所用的代码就已经很费解了，长除法则更加恐怖。想证明这些算法，得绞尽脑汁，光是向上取整和向下取整问题就令人发狂。要写好除法代码，还得跟高德纳这样的大师学，然而就算把神仙请来，恐怕也没办法除以 0 呀！——译者注

第 11 章

初等函数

11.1 整数平方根

“整数平方根”（integer square root）函数是指$\lfloor\sqrt{x}\rfloor$。为扩大其使用范围，同时避免处理负参数，我们假设 x 为无符号数。因此，$0\leqslant x\leqslant 2^{32}-1$。

11.1.1 用牛顿法开平方

浮点数开方时一般都用牛顿法。首先，设法估算出$\sqrt{a}$的初始值 g_0。然后，反复套用下式，即可得到一系列越来越精确的开方值了。

$$g_{n+1}=\left(g_n+\frac{a}{g_n}\right)/2$$

此迭代式呈平方级收敛（converge quadratically，亦称二次收敛），也就是说，如果某个 g_n值精确到了 n 位，则其下一个值 g_{n+1}就会精确到 $2n$ 位。要使用此方法，必须判明迭代次数是否足够，以便终止计算。

令人惊喜的是，用牛顿法对整数开方效果很好。为了理解其原理，需要用到如下定理：

定理 设 $g_{n+1}=\lfloor(g_n+\lfloor a/g_n\rfloor)/2\rfloor$，其中 g_n、a 均为正整数。则有：

(a) 若 $g_n>\lfloor\sqrt{a}\rfloor$，则$\lfloor\sqrt{a}\rfloor\leqslant g_{n+1}<g_n$，

(b) 若 $g_n=\lfloor\sqrt{a}\rfloor$，则$\lfloor\sqrt{a}\rfloor\leqslant g_{n+1}\leqslant\lfloor\sqrt{a}\rfloor+1$。

这就是说，如果用整数 g_n把$\lfloor\sqrt{a}\rfloor$猜得太大了，那么下一次的猜测值 g_{n+1}必定会比这次小，而且大于等于$\lfloor\sqrt{a}\rfloor$。因此，如果初始猜测值比正确值高，那么数列会呈现出单调收敛（converge monotonically）。如果猜测值 g_n等于$\lfloor\sqrt{a}\rfloor$，那么下一个猜测值要么和 g_n相等，要么比它大 1。于是很容易就能判定数列何时收敛：如果以 $g_0\geqslant\lfloor\sqrt{a}\rfloor$开始，那么当 $g_{n+1}\geqslant g_n$时，数列收敛，而 g_n就是正确结果。

$a=0$ 必须特殊处理，因为如果用这个办法计算，就会导致 0 除以 0。

证明：(a) 由于 g_n是整数，所以

$$g_{n+1}=\left\lfloor\left(g_n+\left\lfloor\frac{a}{g_n}\right\rfloor\right)/2\right\rfloor=\left\lfloor\left\lfloor g_n+\frac{a}{g_n}\right\rfloor/2\right\rfloor=\left\lfloor\left(g_n+\frac{a}{g_n}\right)/2\right\rfloor=\left\lfloor\frac{g_n^2+a}{2g_n}\right\rfloor$$

因为 $g_n>\lfloor\sqrt{a}\rfloor$且 g_n是整数，所以 $g_n>\sqrt{a}$。由 $g_n=(1+\varepsilon)\sqrt{a}$来定义ε。由此可知ε>0，并且：

$$\left\lfloor\frac{g_n^2+a}{2g_n}\right\rfloor=g_{n+1}\leqslant\frac{g_n^2+a}{2g_n},$$

$$\left\lfloor\frac{(1+\varepsilon)^2a+a}{2(1+\varepsilon)\sqrt{a}}\right\rfloor=g_{n+1}<\frac{g_n^2+g_n^2}{2g_n},$$

$$\left\lfloor\frac{2+2\varepsilon+\varepsilon^2}{2(1+\varepsilon)}\sqrt{a}\right\rfloor=g_{n+1}<g_n,$$

$$\left\lfloor\frac{2+2\varepsilon}{2(1+\varepsilon)}\sqrt{a}\right\rfloor\leqslant g_{n+1}<g_n,$$

$$\lfloor\sqrt{a}\rfloor\leqslant g_{n+1}<g_n。$$

(b) 因为 $g_n=\lfloor\sqrt{a}\rfloor$，所以$\sqrt{a}-1<g_n\leqslant\sqrt{a}$，于是可知 $g_n^2\leqslant a<(g_n+1)^2$。由此得出

$$\left\lfloor\frac{g_n^2+g_n^2}{2g_n}\right\rfloor\leqslant g_{n+1}\leqslant\left\lfloor\frac{g_n^2+(g_n+1)^2}{2g_n}\right\rfloor,$$

$$\lfloor g_n\rfloor\leqslant g_{n+1}\leqslant\left\lfloor g_n+1+\frac{1}{2g_n}\right\rfloor,$$

$$\lfloor\sqrt{a}\rfloor\leqslant g_{n+1}\leqslant\lfloor g_n+1\rfloor\left(\text{由于 } g_n \text{ 是整数且}\frac{1}{2g_n}<1\right),$$

$$\lfloor\sqrt{a}\rfloor\leqslant g_{n+1}\leqslant\lfloor g_n\rfloor+1=\lfloor\sqrt{a}\rfloor+1。$$

用牛顿法计算$\lfloor\sqrt{x}\rfloor$的难点在于首次估值。图 11.1 将首次猜测值 g_0设置成大于等于$\sqrt{x}$且值最小的 2 的幂。例如，当 $x=4$ 时 $g_0=2$，当 $x=5$ 时 $g_0=4$。

由于首次猜测值是 2 的幂，所以无需真的执行除法操作，只需右移 1 位即可由此算出 g_1。

因为首轮猜测的精确度约是 1 位，而牛顿法呈平方级收敛（也就是说，每一轮的精确位数都是上一轮的两倍），所以大家可能觉得（在 32 位计算机上）5 次迭代就会收敛，于是，共需 4 次除法（首次迭代没用除法，改用右移了）。将所有可能取值都试验过之后，发现最多需执行 5 次除法，或者说，若参数不超过 16 785 407，则最多 4 次。

如果能用前导 0 计数指令，那么首轮猜测值就好算多了：用下列代码替换前 7 行可执行代码即可：

```
if (x <= 1) return x;
s = 16 - nlz(x - 1)/2;
```

```
int isqrt(unsigned x) {
   unsigned x1;
   int s, g0, g1;

   if (x <= 1) return x;
   s = 1;
   x1 = x - 1;
   if (x1 > 65535) {s = s + 8; x1 = x1 >> 16;}
   if (x1 > 255)   {s = s + 4; x1 = x1 >> 8;}
   if (x1 > 15)    {s = s + 2; x1 = x1 >> 4;}
   if (x1 > 3)     {s = s + 1;}

   g0 = 1 << s;                   // g0 = 2**s.
   g1 = (g0 + (x >> s)) >> 1;     // g1 = (g0 + x/g0)/2.

   while (g1 < g0) {              // Do while approximations
      g0 = g1;                    // strictly decrease.
      g1 = (g0 + (x/g0)) >> 1;
   }
   return g0;
}
```

图 11.1 用牛顿法求整数平方根

要是没有前导 0 计数指令，那么还有个办法，就是用二叉查找树计算 s。这种方法算出来的 g_0 要稍好些，它是大于等于 $\lfloor\sqrt{x}\rfloor$ 且值最小的 2 的幂。对于某些 x 值来说，这样求出的 g_0 较小，不过仍然大到符合定理所述的收敛判别标准。下表对比了两种算法。

图 11.1 中 x 的取值范围	图 11.2 中 x 的取值范围	初始猜测值 g_0
0	0	0
1	1～3	1
2～4	4～8	2
5～16	9～24	4
17～64	25～80	8
65～256	81～288	16
……	……	……
$2^{28}+1$～2^{30}	$(2^{14}+1)^2$～$(2^{15}+1)^2-1$	2^{15}
$2^{30}+1$～$2^{32}-1$	$(2^{15}+1)^2$～$2^{32}-1$	2^{16}

图 11.2 列出了这种算法。较小的 x（也就是当 $0\leqslant x\leqslant 24$ 时）按特例处理较为方便，这样就无需计算除法了。

在基本 RISC 架构中，图 11.1 所述算法的最长执行时间约为 26+（D+6）n 个周期，其中 D 是除法所耗周期数，n 是 while 循环执行次数。假设两种算法的分支指令都只需 1 个周期，则图 11.2 在最坏情况下需执行 27+（D+6）n 个周期。下表列出了当 x 均匀分布于相关范围时，这两种算法平均要执行多少次循环。

x	图 11.1	图 11.2
0～9	0.80	0
0～99	1.46	0.83
0～999	1.58	1.44
0～9999	2.13	2.06
0～$2^{32}-1$	2.97	2.97

```
int isqrt(unsigned x) {
   int s, g0, g1;

   if (x <= 4224)
      if (x <= 24)
         if (x <= 3) return (x + 3) >> 2;
         else if (x <= 8) return 2;
         else return (x >> 4) + 3;
      else if (x <= 288)
         if (x <= 80) s = 3; else s = 4;
      else if (x <= 1088) s = 5; else s = 6;
   else if (x <= 1025*1025 - 1)
      if (x <= 257*257 - 1)
         if (x <= 129*129 - 1) s = 7; else s = 8;
      else if (x <= 513*513 - 1) s = 9; else s = 10;
   else if (x <= 4097*4097 - 1)
      if (x <= 2049*2049 - 1) s = 11; else s = 12;
   else if (x <= 16385*16385 - 1)
      if (x <= 8193*8193 - 1) s = 13; else s = 14;
   else if (x <= 32769*32769 - 1) s = 15; else s = 16;
   g0 = 1 << s;              // g0 = 2**s.

   // Continue as in Figure 11-1.
```

图 11.2 用二叉查找树算法求出整数平方根的首轮猜测值

假设除法需要 20 个周期，而 x 均匀分布在 0 至 9999 之间，那么两种算法都需执行 81 个周期。

11.1.2 二分查找

既然牛顿法的首轮猜测值可用二分查找法算出，那为何不把整个计算过程都改用二分查找法呢？此方法首先设置两个边界，其值或许是 0 和 2^{16}。每次猜测值都是两个边界的中点。若中点的平方大于参数 x，则把上界移至中点。若中点的平方小于参数 x，则把下界移至中点。当上下界差 1 时，算法终止，下界就是计算结果。

这个办法无需除法，不过要多次执行乘法，若用 0 和 2^{16} 做初始边界，则需 16 次。（每轮迭代的精确度都比上轮多 1 位。）图 11.3 列出了该算法的变体，其初始边界值取得比 0 和 2^{16} 稍好些。在大多数 RISC 架构的计算机上，图 11.3 的循环可以节省一个周期，它修改了 a 与 b 的值，使得比较语句由原来的 $b-a\geqslant 1$ 变为现在的 $b\geqslant a$。

每次迭代前必须保证 $a\leqslant\lfloor\sqrt{x}\rfloor+1$ 和 $b\geqslant\lfloor\sqrt{x}\rfloor$ 两个谓词成立。b 的初始值应该容易计算，而且接近 $\lfloor\sqrt{x}\rfloor$。合理的初始值是：x，$x\div 4+1$，$x\div 8+2$，$x\div 16+4$，$x\div 32+8$，$x\div 64+16$，等等。刚开始的几个值适合较小的 x，最后几个值适合较大的 x。（以 $x\div 2+1$ 为边界值也可以，但或许不太有用，因为按 $x\div 4+1$ 算出来的边界值不比它差，甚至更好。）

图 11.3 还有 7 种变体。把 a 替换为 $a+1$，把 b 替换为 $b-1$，把 $m=(a+b)\div 2$ 改为 $m=(a+b+1)\div 2$，以上 3 种改法可单独使用，也可结合使用，这样就能生成 7 种变体了。

图 11.3 的执行时间约为 $6+(M+7.5)n$，其中 M 是乘法所需周期数，n 是循环执行次数。下表列出了当 x 均匀分布在相关范围内时循环的平均执行次数。

```
int isqrt(unsigned x) {
   unsigned a, b, m;            // Limits and midpoint.

   a = 1;
   b = (x >> 5) + 8;            // See text.
   if (b > 65535) b = 65535;
   do {
      m = (a + b) >> 1;
      if (m*m > x) b = m - 1;
      else         a = m + 1;
   } while (b >= a);
   return a - 1;
}
```

图 11.3 用简单的二分查找法计算整数平方根

x	循环平均执行次数
0～9	3.00
0～99	3.15
0～999	4.68
0～9999	7.04
$0\sim2^{32}-1$	16.00

若假设乘法耗费 5 个周期，而 x 均匀分布于 0 至 9999 之间，那么算法大约需要 94 个周期。最大执行时间（当 $n=16$ 时）约为 206 个周期。

如果用前导 0 计数指令，那么初始边界就是：

```
b = (1 << (33 - nlz(x))/2) - 1;
a = (b + 3)/2;
```

也就是说，$b=2^{(33-\mathrm{nlz}(x))\overset{u}{\div}2}-1$。对于 x 值较小的情况来说，这样算出来的边界值效果很好（在 $0\leqslant x\leqslant 15$ 时，只需 1 次迭代），但是若 x 值较大，则按图 11.3 算出来的边界值对算法的改善有限。当 x 在 0 至 9999 范围内时，平均迭代次数是 5.45 次，按照与刚才相同的假定，这需要 74 个周期。

11.1.3 硬件算法

9.4.1 节的图 9.2 讲了一种硬件除法算法，与之相似，也可以用这种“移位并相减”的办法来计算平方根。在 32 位计算机上实现时，此算法用了一个 64 位寄存器，把前 32 位初始化为 0，后面跟着参数 x。每次迭代时，64 位寄存器左移两位，当前结果 y（一开始是 0）左移 1 位。然后，用 64 位寄存器的左半边减去 $2y+1$。如果减法结果非负，那么就用其值替换掉 64 位寄存器的左半边，并给 y 加 1（因为此时 y 以 0 结尾，所以不需要用加法器）。如果减法结果为负，那么 64 位寄存器和 y 值均不变。整个过程需要 16 次迭代。

此算法在 1945 年由［JVN］描述。

或许令人惊讶的是，此算法运行时间约是 64÷32⇒32 硬件除法算法的一半，这是因为其迭代次数为除法算法的一半，而两种算法在每轮迭代内的复杂程度差不多。

如用软件编写此算法，那么最好不要使用双字移位寄存器（doubleword shift register），那样做需要大约4条指令才能实现移位。图11.4所列算法［GIS1］通过对 y 与掩码位 m 右移来完成此操作，它平均需要执行149条基本RISC指令。两个 y|m 表达式也可写成 y+m。

```
int isqrt(unsigned x) {
   unsigned m, y, b;

   m = 0x40000000;
   y = 0;
   while(m != 0) {              // Do 16 times.
      b = y | m;
      y = y >> 1;
      if (x >= b) {
         x = x - b;
         y = y | m;
      }
      m = m >> 2;
   }
   return y;
}
```

图 11.4　用硬件算法求整数平方根

此算法的执行过程与小学里用的竖式法很像。下面演示在8位计算机上计算 $\lfloor\sqrt{179}\rfloor$。

```
 1011 0011   x0   Initially, x = 179 (0xB3).
- 1          b1
 ---------
 0111 0011   x1   0100 0000   y1
- 101        b2   0010 0000   y2
 ---------
 0010 0011   x2   0011 0000   y2
-  11 01     b3   0001 1000   y3
 ---------
 0010 0011   x3   0001 1000   y3   (Can't subtract).
-   1 1001   b4   0000 1100   y4
 ---------
 0000 1010   x4   0000 1101   y4
```

这样求出的结果是13，寄存器 x 里剩下10。

可以用常见的“带符号右移31位”（shift right signed 31）这一技巧来取代条件判断语句 if x>=b。可以证明，b的高权重位总是0（实际上 $b \leqslant 5 \cdot 2^{28}$），于是就简化了 x>=b 这一谓词的求值过程（参见2.12节）。简化结果是，把 if 语句块替换为下列代码：

```
t = (int)(x | ~(x - b)) >> 31;   // -1 if x >= b, else 0.
x = x - (b & t);
y = y | (m & t);
```

如果计算机有“or not”（和另一个数的补值取或）指令，那么这样做就会把平均需要3条指令的代码块替换为7条指令，然而，若条件分支语句在此环境下的执行时间大于5个周期，则此办法值得一试。

以软件方式计算整数平方根也许用不着耗费上百个周期吧？似乎应该有简单些的办法才对。有鉴于此，本书提供表11.1中的表达式，用以计算值非常小的参数。如果提前就知道参数比较小的话，那么可以用这些表达式提升上述算法的速度。

表 11.1 基本 RISC 指令集

表达式	适用范围	消费的完全 RICS 指令数
x	0～1	0
$x>0$	0～3	1
$(x+3)\overset{u}{\div}4$	0～3	2
$x\overset{u}{\gg}(x\overset{u}{\div}2)$	0～3	2
$x\overset{u}{\gg}(x>1)$	0～5	2
$(x+12)\overset{u}{\div}8$	1～8	2
$(x+15)\overset{u}{\div}8$	4～15	2
$(x>0)+(x>3)$	0～8	3
$(x>0)+(x>3)+(x>8)$	0～15	5

Ah, the elusive square root,
It should be a cinch to compute.

But the best we can do
Is use powers of two
And iterate the method of Newt![1]

开方虽小实冗繁，
貌似容易算来难。

劝君记取二之幂，
牛顿迭代可补完。

11.2 整数立方根

用牛顿法求立方根效果不太好。迭代公式有点复杂：

$$x_{n+1}=\frac{1}{3}\left(2x_n+\frac{a}{x_n^2}\right)$$

而且猜测初始值 x_0 显然很成问题。

不过，有个硬件算法和开平方用的硬件算法相似，而且用软件实现起来也算不太坏。图 11.5 列出了代码。

因为被加数是偶数，所以其中 3 个“加 1”操作都可改为“和 1 取或”。但即便作如此修改，该算法是否值得用硬件来实现依然值得怀疑，主要原因在于 **y** * (**y**+1) 这个乘法。

使用“复杂运算化简”(strength reduction) 这一编译器优化技术很容易就能省去乘法操作。再引入一个名为 y2 的无符号变量，令其表示 y 的平方，一旦 y 值更新，则 y2 也随之更新。在 y = 0 之前插入一句 y2 = 0。在 y = 2 * y 之前插入一句 y2 = 4 * y2。把对 b 的赋值语句改为 b = (3 * y2 + 3 * y + 1)≪s（并且把因数 3 提取出来）。在 y = y + 1 之前插入一句 y2 = y2 + 2 * y + 1。修改后的程序没有复杂的乘法操作，只有一些小常数乘法，而那些乘法可用移位及加法指令取代。其中 3 次“加 1”操作都可以改为“和 1 取或”。除非

[1] 原诗大意为：平方根真琢磨不透，本来觉得挺容易，谁知算起来这么难。要说最好的办法嘛，那还是先猜一个 2 的幂，然后用牛顿法去迭代吧！——译者注

计算机能在两周期内执行完乘法指令，否则修改后的算法就比之前快。

```
int icbrt(unsigned x) {
   int s;
   unsigned y, b;

   y = 0;
   for (s = 30; s >= 0; s = s - 3) {
      y = 2*y;
      b = (3*y*(y + 1) + 1) << s;
      if (x >= b) {
         x = x - b;
         y = y + 1;
      }
   }
   return y;
}
```

图 11.5　用硬件算法计算整数立方根

注意事项：[GLS1] 指出，如果把图 11.5 的代码及运用了“复杂运算化简”技术之后的变种照搬到 64 位计算机上，那么算法就无效了。对 b 的赋值可能会溢出。要解决此问题，可以把“左移 s 位”这一操作从对 b 的赋值语句中拿掉，在赋值语句后面加一句 bs = b≪s，并且把“if(x> = b){x = x - b...”这两行改为“if(x> = bs && b = = (bs≫s)){x = x - bs...”。

11.3　求整数幂

11.3.1　用 n 的二进制分解式计算 x^n

如果 n 为非负整数，那么计算 x^n 的常用办法就是将 n 进行二进制分解。只要表达式具备 $x \cdot x \cdot x \cdot \ldots \cdot x$ 的形式，就可以用此技术求值，其中的 · 可以是任何一种“可结合操作”(associative operation)，例如加法、乘法（含矩阵乘法）、字符串拼接（正如记法（‘ab’)^3 = ‘ababab’ 所示，字符串拼接也是一种“可结合操作”）。比方说，想计算 $y = x^{13}$ 的值。因为 13 可表示为二进制 1101（也就是说 13=8+4+1），所以

$$x^{13} = x^{8+4+1} = x^8 \cdot x^4 \cdot x^1$$

于是，可按如下步骤计算 x^{13}：

$$t_1 \leftarrow x^2$$
$$t_2 \leftarrow t_1^2$$
$$t_3 \leftarrow t_2^2$$
$$y \leftarrow t_3 \cdot t_2 \cdot x$$

这需要 5 条乘法指令，比其连乘 x 所需的 12 条要少很多。

如果已知指数是非负整数，那么可以把此技巧总结为图 11.6 这样的子程序。

当指数 $n \geqslant 1$ 时，此方法要执行的乘法次数为

$$\lfloor \log_2 n \rfloor + nbits\ (n) - 1$$

这样算并不总是能把乘法次数减至最低。例如，当时 $n=27$ 时，二进制分解式为

$$x^{16} \cdot x^8 \cdot x^2 \cdot x^1$$

这需要 7 次乘法。然而下述方法只需 6 次：

$$((x^3)^3)^3$$

用二进制分解法无法得出最优方案的最小 n 值是 15（提示：$x^{15}=(x^3)^5$）。

也许令人惊讶的是，没有哪种简单办法能够找到计算 x^n 所用的最佳乘法序列。目前所知的唯一方案要用执行“大量的搜索”（extensive search）才行。[Knu2，4.6.3] 详细论述了此问题。

二进制分解法还有个变体，就是把指数表示成二进制，由左至右扫描 [Rib，32]，此法与由左至右将二进制数转换为十进制数相似。将结果 y 初始为 1，由左至右扫描指数。如果遇见 0，就对 y 取平方。如果遇见 1，则先把 y 取平方，再与 x 相乘。以此法计算 $x^{13}=x^{1101_2}$，就是

$$(((1^2 \cdot x)^2 \cdot x)^2)^2 \cdot x$$

和图 11.6 那种从右至左的算法相比，此算法要执行的乘法次数与之相同，这两种情况下的乘法的操作数都不小（nontrivial）[⊖]。

```
int iexp(int x, unsigned n) {
   int p, y;

   y = 1;                      // Initialize result
   p = x;                      // and p.
   while(1) {
      if (n & 1) y = p*y;      // If n is odd, mult by p.
      n = n >> 1;              // Position next bit of n.
      if (n == 0) return y;    // If no more bits in n.
      p = p*p;                 // Power for next bit of n.
   }
}
```

图 11.6 用 n 的二进制分解式计算 x^n

11.3.2 用 Fortran 语言计算 2^n

IBM XL Fortran 编译器将 pow2（n）函数定义为：

$$\text{pow2}\ (n) = \begin{cases} 2^n, & 0 \leqslant n \leqslant 30, \\ -2^{31}, & n=31, \\ 0, & n<0 \text{ 或 } n \geqslant 32。\end{cases}$$

函数假设 n 与计算结果均是带符号数。ANSI/ISO Fortran 标准要求当 $n<0$ 时结果是 0。如果把 $n \geqslant 31$ 时的函数值看成正确结果除以 2^{32} 后的余数，那么此情况下的函数定义还算合理，而且也与连乘得出的结果相符。

⊖ 原书把这种乘法叫做“非平凡的乘法”（nontrivial multiplication）。——译者注

计算 2^n 的标准方式是把整数1放在寄存器里，然后左移 n 位。这不符合 Fortran 的定义，因为移位量通常要和64或32取模（在32位计算机上），于是将导致移位量过大或为负数时结果错误。

如果计算机有前导0计数指令，可以按如下方式，以4条指令算出 pow2(n)[Shep]：

$$
\begin{aligned}
&\mathbf{x} \leftarrow \mathbf{nlz}(\boldsymbol{n} \overset{u}{\gg} \mathbf{5}); && \text{//如果 } 0 \leqslant n \leqslant 31\text{，则 } x \leftarrow 32\text{，否则 } x \leftarrow x < 32\text{。}\\
&\mathbf{x} \leftarrow \mathbf{x} \overset{u}{\gg} \mathbf{5}; && \text{//如果 } 0 \leqslant n \leqslant 31\text{，则 } x \leftarrow 1\text{，否则 } x \leftarrow 0\text{。}\\
&\mathbf{pow2} \leftarrow \mathbf{x} \ll \boldsymbol{n};
\end{aligned}
$$

右移位操作是“逻辑的”(logical)，即便 n 是带符号数也不传播符号。

如果计算机没有 nlz 指令，那么上面用到该指令的地方就可以替换为2.12节计算命令谓词 $\mathbf{x}=\mathbf{0}$ 所用的程序，并把表达式 $\mathbf{x} \overset{u}{\gg} \mathbf{5}$ 改为 $\mathbf{x} \overset{u}{\gg} \mathbf{31}$。还有个办法也需更好：由于 $\mathbf{0} \leqslant \mathbf{x} \leqslant \mathbf{31}$ 这个谓词与 $\mathbf{x} \overset{u}{<} \mathbf{32}$ 等价，所以可根据2.12节计算命令谓词 $\mathbf{x} \overset{u}{<} \mathbf{y}$ 所用的代码，把此表达式化简为 $\neg \mathbf{x}\ \&\ (\mathbf{x}-\mathbf{32})$。这样只需5条指令（如果计算机有“and not”指令，就是4条）：

$$
\begin{aligned}
&\mathbf{x} \leftarrow \neg\, \boldsymbol{n}\ \&(\boldsymbol{n}-\mathbf{32}); && \text{//当且仅当 } 0 \leqslant n \leqslant 31 \text{ 时，} x<0\text{。}\\
&\mathbf{x} \leftarrow \mathbf{x} \overset{u}{\gg} 31; && \text{//当 } 0 \leqslant n \leqslant 31 \text{ 时，} x=1\text{，否则 } x=0\text{。}\\
&pow2 \leftarrow \mathbf{x} \ll \boldsymbol{n};
\end{aligned}
$$

11.4 整数对数

“整数对数”(integer logarithm) 函数是指函数 $\lfloor \log_b x \rfloor$，其中 x 是正整数，而 b 是大于等于2的整数。通常 b 是2或10，这两种函数分别叫做“ilog2”、“ilog10”。如果未指明底数，就称为“ilog”。

如果定义 ilog(0)$=-1$，那么就能把定义域拓展至 $x=0$ 了[CJS]。这样定义有几个原因：

- 如果把 $x=0$ 的情况包含在内，那么按照下述公式，很容易就能把 ilog2(x) 函数和前导0计数函数 nlz(x) 联系起来。

  $$\mathrm{ilog2}(x)=31-\mathrm{nlz}(x)$$

- 如果这样定义的话，那么很容易就能根据下述公式算出 $\lceil \log(x) \rceil$。当 $x=1$ 时，此公式意味着 ilog(0)$=-1$。

  $$\lceil \log(x) \rceil=\mathrm{ilog}(x-1)+1$$

- 如果这样定义的话，那么下列等式在 $x=1$ 时就可以成立了（不过在 $x=0$ 时不成立）。

  $$\mathrm{ilog2}(x \div 2)=\mathrm{ilog2}(x)-1$$

- 如果这样定义的话，那么 ilog(x) 的结果就构成一个小的稠密整数集（在 32 位计算机上，如果 x 为无符号数，则 ilog2(x) 的运算结果是 -1 至 31），其值可用作下标直接查表。
- 从很多计算 ilog2(x) 和 ilog10(x) 的算法中都可得出这一结果。

不巧的是，此函数不等于"x 的二进制位数"，那个值应该是 ilog(x)$+1$($x=0$ 除外)。最好把 $x=0$ 看成特例。

当 $x<0$ 时，ilog(x) 未定义。为扩大其使用范围，我们把函数定义为从无符号数到带符号数的映射。这样一来，就不会出现负参数了。

11.4.1 以 2 为底的整数对数

计算 ilog2(x) 和 5.3 节所述的计算前导 0 个数在本质上是一样的。很容易就能把那一节所说的各种算法直接改编为 ilog(2) 函数，而无需先计算 nlz(x)，然后再从 31 中减去。（对于 5.3 节的图 5.16 来说，把 `return pop(~x)` 这一行改为 `return pop(x) - 1`。）

11.4.2 以 10 为底的整数对数

有时需要把二进制数转换成十进制数，并去掉其前导 0，以便将多个十进制数压缩到一行，在这个应用场景中可以使用 ilog10 函数。转换过程中要反复除以 10，先算最低有效位。如果提前知道最低有效位在哪，那么就不用把转换好的数字先放在临时区域然后再移动了。

用"表格搜索法"（table search）来计算 ilog10(x) 比较合适。可以使用二分查找法，不过由于表格较小，而且许多应用场景中的 x 通常也较小，所以简单的线性搜索也许最合适。图 11.7 列出了这个相当直观的程序。

在基本 RISC 架构的计算机上，此程序大概要执行 $9+4\lfloor \log_{10}x \rfloor$条指令。因此，该算法要执行 5 至 45 条指令，在典型情况下（当 $10\leqslant x\leqslant 99$ 时）也许是 13 条。

很容易就能把图 11.7 改编为"寄存器内"版本（"in register" version，不用表格）。图 11.8 列出了此种算法的可执行部分。要是计算机能快速和 10 相乘的话，那么这种办法也许有用。

```
int ilog10(unsigned x) {
   int i;
   static unsigned table[11] = {0, 9, 99, 999, 9999,
      99999, 999999, 9999999, 99999999, 999999999,
      0xFFFFFFFF};

   for (i = -1; ; i++) {
      if (x <= table[i+1]) return i;
   }
}
```

图 11.7 用简单的表格搜索法计算以 10 为底的整数对数

```
p = 1;
for (i = -1; i <= 8; i++) {
   if (x < p) return i;
   p = 10*p;
}
return i;
```

图 11.8 用反复乘以 10 的办法计算以 10 为底的整数对数

如果把乘法算作1条指令，那么在基本RISC架构的计算机上，此程序大约执行$10+6\lfloor \log_{10}x \rfloor$条指令。当$10 \leqslant x \leqslant 99$时，需要16条。

若用二分查找法，则可实现出无循环且不用表的算法。这种算法可以先把x和10^4比较，然后再同10^2或10^6比较，以此类推，直至找到符合$10^n \leqslant x < 10^{n+1}$的指数$n$为止。沿着此算法的路径执行，需要10至18条指令，其中4、5条是分支指令（算上最后那条无条件分支语句）。

图11.9是改版后的二分查找法，不论沿着哪条路径执行，最多只需4个分支，此算法对值较小的x有利。当$10 \leqslant x \leqslant 99$时，需要执行6条基本RISC指令，当$x \geqslant 100$时，需要11至16条。

```
int ilog10(unsigned x) {
   if (x > 99)
      if (x < 1000000)
         if (x < 10000)
            return 3 + ((int)(x - 1000) >> 31);
         else
            return 5 + ((int)(x - 100000) >> 31);
      else
         if (x < 100000000)
            return 7 + ((int)(x - 10000000) >> 31);
         else
            return 9 + ((int)((x-1000000000)&~x) >> 31);
   else
      if (x > 9) return 1;
      else       return ((int)(x - 1) >> 31);
}
```

图11.9　用修改后的二分查找法计算以10为底的整数对数

程序中的移位指令是带符号移位（所以才要用（int）来强制转换类型）。如果计算机没有此指令，那么可在下面列出的替代方案里任选一种，这些都使用无符号移位。它们都以第一条return语句为例。不巧的是，头两种方案都需要高效实现的立即数减法（subtract form immediate）指令，而大多数计算机做不到。最后一种方案要与大常数相加（需要两条指令），然而这对第二和第三个return语句来说不是问题，因为这两条语句无论如何都得和大常数相加。最大的常数是$2^{31}-1000$。

```
return 3 - ((x - 1000) >> 31);
return 2 + ((999 - x) >> 31);
return 2 + ((x + 2147482648) >> 31);
```

第四个return语句还可以写成：

```
return 8 + ((x + 1147483648) | x) >> 31;
```

其中的大常数是$2^{31}-10^9$。这么写就不需要“and not”和带符号移位指令了。

可用如下两条语句之一来替换最后那个if-else语句块，这样能省一个分支：

```
return ((int)(x - 1) >> 31) | ((unsigned)(9 - x) >> 31);
return (x > 9) + (x > 0) - 1;
```

如果nlz(x)或ilog2(x)可用一条指令来实现，那么就能以更好、更有意思的办法来

算 ilog10(x) 了。例如，图 11.10 的代码用两次查表法［CJS］来计算此值。

table1 已经估算出 ilog10(x) 的大概值了。这个数值一般来说就是正确答案，不过在 $x=0$ 以及 x 位于 8～9、64～99、512～999、8192～9999 等区间时，估算值会比正确值大 1。第二张表里列出了一些数字，如果估算值比相应数字小，那么就要减 1。

此方案用了长度为 73 字节的表，在 IBM System/370 计算机上只需 6 条指令（这么做的前提是：table1 中的值必须是下列代码清单中的 4 倍）。在具备前导 0 计数指令的 RISC 架构计算机上，大约要执行 10 条指令。接下来要讨论的其他方法都是此方法的变体。

第一种变体是把 if 语句的条件分支去掉。实际上，如果计算机有"set less than unsigned"（如果一个无符号数比另一个小，则把目标寄存器置 1，简称 SLTU）指令，那么就可以把图 11.10 改成无分支代码，不过，接下来要说的这个变种方法可在不具备特殊指令（unusual instruction，前导 0 计数指令在这里算作常规指令）的计算机上用。

```
int ilog10(unsigned x) {
   int y;
   static unsigned char table1[33] = {9, 9, 9, 8, 8, 8,
      7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3,
      2, 2, 2, 1, 1, 1, 0, 0, 0, 0};
   static unsigned table2[10] = {1, 10, 100, 1000, 10000,
      100000, 1000000, 10000000, 100000000, 1000000000};

   y = table1[nlz(x)];
   if (x < table2[y]) y = y - 1;
   return y;
}
```

图 11.10 通过两次查表，根据以 2 为底的对数算出以 10 为底的整数对数

这个方法是，把 if 语句换成减法，后面再跟一条右移 31 位的指令，这样就可以把符号位从 y 中减掉了。当 x 较大时（$x \geqslant 2^{31}+10^{9}$）比较难算，这种情况下可以再和 table2 中的元素相加，如图 11.11 所示。

在一台除了前导 0 计数指令外差不多都是基本指令的 RISC 计算机上，此算法需要执行 11 条指令。在 $x=0$ 时，也可以把 table1 最后一个元素改为 1（也就是把表格最后的"0，0，0，0"改成"0，0，0，1"），这样返回值就从 -1 变成 0 了（在应对十进制数转换问题时，这么改比较好）。

还有一种变体，用减法、乘法、移位来取代第一次查表操作。这看上去似乎可行，因为 $\log_{10}x$ 等于 $\log_2 x$ 乘以一个常数，这个常数就是 $\log_{10}2$，其值是 0.301 03…。因此，只要 c 值取得合适（约为 0.301 03），应该就能计算出 ilog10(x) 了：先算 $\lfloor c\ \mathrm{ilog2}(x) \rfloor$，然后根据图 11.11 中像 table2 这样的修正表格来调整其值。

为了实现此算法，设 $\log_{10}2=c+\varepsilon$，其中 c 是大于 0 且比较接近 $\log_{10}2$ 的数，用它做乘数很方便，而 $\varepsilon>0$。于是，当 $x \geqslant 1$ 时：

$$\mathrm{ilog10}(x)=\lfloor \log_{10}x \rfloor=\lfloor (c+\varepsilon)\log_2 x \rfloor$$

$$\lfloor c\log_2 x \rfloor \leqslant \mathrm{ilog10}(x)=\lfloor c\log_2 x+\varepsilon\log_2 x \rfloor$$

$$\begin{aligned}\lfloor c\,\mathrm{ilog2}(x)\rfloor \leqslant \mathrm{ilog10}(x) &\leqslant \lfloor c(\mathrm{ilog2}(x)+1)+\varepsilon\log_2 x\rfloor\\ &\leqslant \lfloor c\,\mathrm{ilog2}(x)+c+\varepsilon\log_2 x\rfloor\\ &\leqslant \lfloor c\,\mathrm{ilog2}(x)\rfloor+\lfloor c+\varepsilon\log_2 x\rfloor+1。\end{aligned}$$

```
int ilog10(unsigned x) {
   int y;
   static unsigned char table1[33] = {10, 9, 9, 8, 8, 8,
      7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3,
      2, 2, 2, 1, 1, 1, 0, 0, 0, 0};
   static unsigned table2[11] = {1, 10, 100, 1000, 10000,
      100000, 1000000, 10000000, 100000000, 1000000000,
      0};

   y = table1[nlz(x)];
   y = y - ((x - table2[y]) >> 31);
   return y;
}
```

图 11.11 用无分支代码查表两次，根据以 2 为底的对数算出以 10 为底的整数对数

因此，如果所选的 c 能令 $c+\varepsilon\log_2 x<1$，那么就可以用$\lfloor c\ \mathrm{ilog2}(x)\rfloor$来估算ilog10($x$)了，估算值要么刚好正确，要么和其差 1。而且，如果令 ilog2(0)＝ilog10(0)＝－1，那么$\lfloor c\ \mathrm{ilog2}(0)\rfloor$＝ilog10(0)（因为 $0<c\leqslant 1$），于是就不用担心这种情况了。（这里使用其他定义也行，例如可规定 ilog2(0)＝ilog10(0)＝0。）

因为 $\varepsilon=\log_{10}2-c$，所以选定的 c 必须满足：

$$c+(\log_{10}2-c)\log_2 x<1,\text{或者说}$$
$$c(\log_2 x-1)>(\log_{10}2)\log_2 x-1。$$

在 x 为 1，2 时，上式必然成立（因为 $c<1$）。如果 x 更大的话，那么就必须令 c 满足

$$c>\frac{(\log_{10}2)\log_2 x-1}{\log_2 x-1}$$

x 越大，对 c 的要求就越严。在 32 位计算机上，$x<2^{32}$，所以选择

$$c>\frac{0.30103.32-1}{32-1}\approx 0.27848$$

就够了。由于 $\varepsilon>0$，所以 $c<0.301\ 03$，于是选取 $c=9/32=0.281\ 25$ 比较方便。试验表明，5/16 和 1/4 这种比较粗略的近似值不可行。

上述思路可总结为图 11.12，此算法若是发现低估了结果，则为其加 1。在具备前导 0 计数指令的 RISC 计算机上，如果把乘法算作 1 条指令，那么大约要执行 11 条指令。

也可以改写为无分支版本，不过处理较大的 x 值时（$x>2^{31}+10^9$）仍会遇到麻烦，这种情况下有两个修正方案。其中一种方案是改用 19/64 做乘数，并把表格扩大一点。图 11.13 演示了这种做法（在具备前导 0 计数指令的 RISC 计算机上，如果把乘法算作 1 条指令，那么大约要 11 条指令）。

另外一种方案是，把减法结果和 x 取“或”，这就等于在 $x\geqslant 2^{31}$ 时强行设置符号位，也就是说，把图 11.12 的第 2 行可执行代码换成

```
y = y + (((table2[y+1] - x) | x) >> 31);
```

要是计算机执行乘 19 要比执行乘 9 慢很多的话（用移位及加法来实现乘法时的确如此），那么这个办法更好。

```
static unsigned table2[10] = {0, 9, 99, 999, 9999,
   99999, 999999, 9999999, 99999999, 999999999};

y = (9*(31 - nlz(x))) >> 5;
if (x > table2[y+1]) y = y + 1;
return y;
```

图 11.12 用一次查表法，根据以 2 为底的对数算出以 10 为底的整数对数

```
int ilog10(unsigned x) {
   int y;
   static unsigned table2[11] = {0, 9, 99, 999, 9999,
      99999, 999999, 9999999, 99999999, 999999999,
      0xFFFFFFFF};

   y = (19*(31 - nlz(x))) >> 6;
   y = y + ((table2[y+1] - x) >> 31);
   return y;
}
```

图 11.13 用无分支的代码查表一次，根据以 2 为底的对数算出以 10 为底的整数对数

在 64 位计算机上，选用下列 c 值就够了：

$$c > \frac{0.30103.64-1}{64-1} \approx 0.28993$$

使用 19/64=0.296 875 较为方便，实验表明，更为粗略的数值不可行。(无分支版的）算法程序如下：

```
unsigned table2[20] = {0, 9, 99, 999, 9999, ...,
   9999999999999999999};
y = ((19*(63 - nlz(x)) >> 6;
y = y + ((table2[y+1] - x) >> 63;
return y;
```

11.5 习题

1. 对 x 的整数平方根再求整数平方根，其结果是否等于 x 的整数四次方根？也就是说，

$$\left\lfloor \sqrt{\lfloor \sqrt{x} \rfloor} \right\rfloor = \left\lfloor \sqrt[4]{x} \right\rfloor \text{是否成立？}$$

2. 将 11.2 节末尾提到的 64 位版立方根算法用代码写出来，使用 C 语言的 “long long” 数据类型。在处理 b 的溢出时，有没有另外一种可能会稍快些的办法？
3. 计算 x^{23} 需要几次乘法（计算结果要和 2^W 取模，其中 W 为计算机字长）？
4. 当 x 为大于 0 的整数时，用简单的语言描述如下两个函数的含义：(a) $2^{\mathrm{ilog2}(x)}$、(b) $2^{\mathrm{ilog2}(x-1)+1}$。

第 12 章

以特殊值为底的数制

本章讨论几个不常见的按位计数系统（positional number system）。这些内容也许并不实用，然而颇有趣味，亦可满足大家的好奇心。我们只讨论整数，不过其原理可扩展到小数点（radix point）后面的数位，该种情况通常表示非整数（non-integer），不过也有例外。

12.1 以−2为底的数制

如果以−2为底，那么正整数和负整数都无需标明正负号，也无需使用令最高有效位具备负权重等非常规手段（[Knu3]）。与+2为底的情况相同，数位也用“0”、“1”两个值；也就是说，包含“0”、“1”的位串含义如下：

$$(a_n\ldots a_3a_2a_1a_0)=a_n(-2)^n+\ldots+a_3(-2)^3+a_2(-2)^2+a_1(-2)+a_0。$$

由此可知，想把某整数化成以−2为底的数，或者说“负二进制”（negabinary）数，可以反复将此数除以−2，并记下余数。这种除法给出的余数必是0或1（也就是“负二进制”表示数位所用的那两个数字），因此，必须执行“余数非负式除法”（modulus division）。举个例子，想用“负二进制”来表示−3，可按如下步骤执行：

$$\frac{-3}{-2}=2\text{ 余 }1$$

$$\frac{2}{-2}=-1\text{ 余 }0$$

$$\frac{-1}{-2}=1\text{ 余 }1$$

$$\frac{1}{-2}=0\text{ 余 }1$$

商变为0之后，就不用再往下算了（若是再往下算，商和余数就都是0了）。于是，从下至上读出余数，可知−3的“负二进制”数是1101。

表12.1左侧两栏列出了从0000至1111的负二进制位串及其对应的十进制数，右侧三栏列出了−15至+15之间的十进制数如何用“负二进制”表示。

表 12.1 在十进制与“负二进制”之间转换

n (负二进制)	n (十进制)	n (十进制)	n (负二进制)	n (负二进制)
0	0	0	0	0
1	1	1	1	11
10	−2	2	110	10
11	−1	3	111	1101
100	4	4	100	1100
101	5	5	101	1111
110	2	6	11010	1110
111	3	7	11011	1001
1000	−8	8	11000	1000
1001	−7	9	11001	1011
1010	−10	10	11110	1010
1011	−9	11	11111	110101
1100	−4	12	11100	110100
1101	−3	13	11101	110111
1110	−6	14	10010	110110
1111	−5	15	10011	110001

n 位字组的 2^n 种位串能否表示某个范围内的全部整数似乎不易直接看出，然而可通过归纳法证明。归纳假设是：用 n 位字组可表示下列范围内的全部整数：

$$\text{当 } n \text{ 为偶数时，} -(2^{n+1}-2)/3 \text{ 至} (2^n-1)/3 \tag{1a}$$

$$\text{当 } n \text{ 为奇数时，} (-(2^n-2)/3) \text{ 至} ((2^{n+1}-1)/3)\text{。} \tag{1b}$$

首先假定 n 是偶数。当 $n=2$ 时，可表示的负二进制数是 10，11，00，01，或者说

$$-2,\ -1,\ 0,\ 1$$

这与 (1a) 相符，而且范围内的每个整数都出现了 1 次，并且仅出现 1 次。

对于 $n+1$ 位的负二进制数，如果其首位是 0，则可表示 (1a) 范围内的全部整数。此外，如果首位是 1，则可表示刚才那个范围再加上 $(-2)^n=2^n$ 之后的所有整数。这个范围是：

$$2^n-\ (2^{n+1}-2)/3 \text{ 至 } 2^n+(2^n-1)/3$$

或者说

$$(2^n-1)/3+1 \text{ 至} (2^{n+2}-1)/3\text{。}$$

此范围与 (1a) 所定义的范围相连，所以，$n+1$ 位的负二进制数可以表示

$$-(2^{n+1}-2)/3 \text{至} (2^{n+2}-1)/3$$

范围内的所有整数，而且每个整数只出现一次。把 (1b) 中的 n 换为 $n+1$，即可知上述范围与 (1b) 相符。

在 n 为奇数时，可以由 (1b) 推出 (1a)，而且可知 (1a) 范围内的所有整数都只有一种表示方法，这个证明过程与 n 为偶数时类似。

加减法的常用规则（如 $0+1=1$ 及 $1-1=0$ 等）当然照样适用。由于 2 可写为 110，

而−1可写为11等原因，所以下述规则也适用。用下面这些规则，连同刚才所说的几条明显规则，足以运算负二进制的加减法了。

$$\begin{aligned} 1+1&=110\\ 11+1&=0\\ 1+1+1&=111\\ 0-1&=11\\ 11-1&=10 \end{aligned}$$

执行加减法时可能出现两个进位。要把进位加到其所在的栏中，即便减法也是如此。在出现两个进位时，可以把它们都放在当前位左侧的那一位上，如有可能，就使用11+1=0来化简。如果“11”进位到包含两个“0”的栏中，则留下一个“1”，把另一个“1”再向左进位。下面举两个例子。

```
             加法                             减法

  11 1 11     11                    1 11  1     1
      1  0  1  1  1    19                 1  0  1  0  1     21
  + 1 1  0  1  0  1 +(-11)           - 1  0  1  1  1  0  -(-38)
  ----------------- ------           ----------------- -----
  0  1  1  0  0  0      8            1  0  0  1  1  1  1    59
```

只可能出现0、1、11这三种进位。如果最高权重位还产生进位的话（不论进位是1还是11），就溢出了。加减法都是如此。

由于进位有三种情况，所以“负二进制加法器”（base-2 adder）要比“二补码加法器”（two's-complement adder）更复杂。

有两种办法可以求出整数的相反数。一种是把整数左移一位，再与原数相加（相当于乘以−1），另一种是用0减去此数。想在“二补码算术”（two's-complement arithmetic）中求相反数，只需“按位取反再加1”（complement and add 1）即可，而负二进制数则没有如此简单的规则。在二补码算术中，通过这条规则，可用加法器实现出减法器（按照 $\boldsymbol{A}+\bar{\boldsymbol{B}}+1$ 来算 $\boldsymbol{A}-\boldsymbol{B}$）。

在底数为−2时，没有那么简单的办法可循，不过有一种办法和二补码算术差不多，也很容易：先把被减数取反（也就是将每个位元的值反转），然后将取反之后的被减数与减数相加，再对二者之和取反［Lang］。下面演示用此方法在8位计算机上计算6减13。

```
00011010      6
00011101      13
11100101      6 取反后的数
------------
11110110      (6 取反后的数) + 13
00001001      对上面两数之和取反 (答案是 - 7)
```

这个办法相当于在负二进制算术使用下列公式，其中 $\boldsymbol{I}$ 代表每个数位均为1的字组：

$$\boldsymbol{A}-\boldsymbol{B}=\boldsymbol{I}-((\boldsymbol{I}-\boldsymbol{A})+\boldsymbol{B})$$

与−2为底的整数相乘很容易。先用1×1=1，0乘以0或1等于0这套规则列乘法

竖式，然后再用负二进制数的加法规则把每一列中的数字累加即可。

然而，除法却相当复杂。想实现一套合理的硬件除法算法（也就是说，通过反复执行减法与移位操作来实现除法）真是项挑战。为明确其意，图 12.1 演示了 8 位计算机上的除法算法。该算法执行“余数非负式除法”（此种除法的余数不会为负）。

```
int divbm2(int n, int d) {           // q = n/d in base -2.
   int r, dw, c, q, i;

   r = n;                            // Init. remainder.
   dw = (-128)*d;                    // Position d.
   c = (-43)*d;                      // Init. comparand.
   if (d > 0) c = c + d;
   q = 0;                            // Init. quotient.
   for (i = 7; i >= 0; i--) {
      if (d > 0 ^ (i&1) == 0 ^ r >= c) {
         q = q | (1 << i);           // Set a quotient bit.
         r = r - dw;                 // Subtract d shifted.
      }
      dw = dw/(-2);                  // Position d.
      if (d > 0) c = c - 2*d;        // Set comparand for
      else c = c + d;                // next iteration.
      c = c/(-2);
   }
   return q;                         // Return quotient in
                                     // base -2.
                                     // Remainder is r,
}                                    // 0 <= r < |d|.
```

图 12.1　负二进制除法

虽说此算法以 C 语言代码编写，且测试它所用的计算机执行的是二补码算术，但是这些都不是实质问题，我们应该抽象地看待它才对。代表输入值的变量 n 与 d 以及除 q 外的其余中间变量都是与进制无关的数而已。输出值 q 是以负二进制表示的位串。

此算法需略加说明。如果输入值以负二进制表示，那么算法就很难以可执行代码来表示了。比方说，如果用负二进制的话，那么“if (d>0)”这行测试代码就必须检测 d 的最高有效位是否位于偶数位置上。“c = c + d”这条加法语句也必须改用负二进制下的规则来写。那样写出来的代码很难读懂。按照本书所列代码来实现算法时，应该把 n 和 d 理解成与进制无关的数。代码只是在演示算术操作的过程，无关其进制。假如代码中的数字与用硬件实现算法时一样都是负二进制的话，那么乘以 −128 就相当于左移 7 位，除以 −2 就相当于右移 1 位。

下面举几个数值来说明代码的运算结果：

divbm2(6, 2) = 7(6除以 2，商为 111_{-2})

divbm2(−4, 3) = 2(负4除以 3，商为 10_{-2})

divbm2(−4, −3) = 6(负4除以负 3，商为 110_{-2})

“q = q |(1≪i);”这一步只是简单地把 q 的第 i 位设置为 1，下面那行“r = r − dw”表示把除数 d 移位，然后将其从余数中减去。

此算法很难详细描述，不过本书尽力讲解其大意。

思考一下如何算出商的首位，也就是q的7号位。在负二进制下，一个8位数的最高有效位如果是1，那么这个数就在-170至-43之间。因此，在不考虑溢出的情况下，当且仅当商的代数值小于等于-43时，其首位（也就是最高有效位）才会是1。

由于$n=qd+r$，而且在除数为正时，$r\leqslant d-1$，所以对正除数来说，当且仅当$n\leqslant -43d+(d-1)$或者说$n<-43d+d$时，商的首位才会是1。对负除数来说，当且仅当$n\geqslant -43d$时（"余数非负式除法"算出的余数r大于等于0），商的首位才会是1。

因此，当且仅当下式成立时，商的首位才会是1：

$$(d>0\ \&\ \neg(n\geqslant -43d+d))\ |\ (d<0\ \&\ n\geqslant -43d)$$

忽略$d=0$的情况，上式可写为：

$$d>0 \oplus n\geqslant c$$

当$d\geqslant 0$时，其中的c是$-43d+d$，当$d<0$时，其中的c是$-43d$。

偶数位置上的商可按此法确定。计算奇数位置上的商时，要把上述逻辑反转。因此，测试条件中要包含"（i & 1）== 0"这一项。（程序代码中的^字符表示异或）

每轮迭代前，都会把c设为绝对值最小且除以d后第i位是1的值。若当前余数r超过该值，则把q的第i位设为1，并调整r，调整方法是：构造一个仅有第i位是"1"的数，将其乘以除数d，再把积从r里减去。此处不用真的执行乘法，只需把d适当移位，然后令r与之相减即可。

算法并不优雅。实现中的难点在于，其中有若干次加法、减法、比较操作，甚至刚开始还要算（乘数是常量的）乘法。也许有人想找"统一"算法（"uniform" algorithm），也就是令算法不要检测参数的正负号，而是根据每步的运算结果来执行不同操作。但是，在负二进制中恐怕无法实现这种统一算法（或者说无法用二补码算术来实现）。原因在于，除法的执行过程本来就不统一。思考一下"移位并相减"（shift-and-subtract）这种最简单的除法算法。这种算法可能根本就不移位，比如在参数为正时，只是反复把除数从被除数中减去，直至余数小于除数为止，此时已经执行的减法总次数就是商。另一方面，如果被除数为负（而除数为正），那么就反复把除数加到被除数上，直至余数为0或正数为止，此时已经执行的加法总次数就是商。若除数为负，则过程又会不同。

虽说如此，但是若用"原码"[⊖]来表示数，则除法过程就统一了。如果使用了这种表示方法，那么由于纯数量本身是正值，所以算法可把这个量反复从被除数中减去，直至余数为负，然后把商的符号位设置成两个参数的异或值，并令余数的符号位与被除数相

⊖ signed-magnitude representation，又称 Sign-and-magnitude representation，此表示法通常用最高有效位来表示正负号，"0"为正，"1"为负，用其他数位来表示纯数量（或者说数的绝对值）。例如在8位带符号原码中，0111 1111 表示十进制数127，1111 1111 表示十进制数-127，这两个8位原码中，最左侧的"0"、"1"分别是其符号位，后面7个"1"是纯数量（相当于十进制的127）。详情参见：http://zh.wikipedia.org/wiki/有符号数处理。——译者注

同（这就是“向 0 取整式除法”的结果）。

在某种意义上讲，上述算法还能做得更统一些：首先看除数是否为负，是则求反，然后按照 $d>0$ 时的简化步骤来执行。执行完整个过程后再修正。“余数非负式除法”的修正步骤就是将商取相反数，而余数不变。这样可以把某些测试移出循环，不过算法整体看起来还是不够理想。

可以把常用的数位表示法（number representation）和负二进制比较一下，看看计算机硬件在执行基本四则运算操作（four fundamental arithmetic operations）时，能否“统一”（uniformly）处理这些数。“统一”这个词没有精确定义，不过基本上就是说，运算过程中不能出现那种需要依参数正负号而酌情执行的操作。把运算结果的符号位设置成两参数正负号的异或值，这也算一种“统一操作”（uniform operation）。表 12.2 列出了在各种数位表示法中这四种基本算术操作能否统一处理其操作数。

通过“首位循环进位”（end around carry）这一技巧，“一补码”（又称反码，One's-complement）的加减法可统一执行。执行加法时，把包括符号位在内的全部位元都按普通二进制数的规则相加，然后把最左侧（也就是符号位）所产生的进位加到最低有效位上。这个过程总会停下来（也就是说，把符号位产生的进位加到最低有效位之后，不可能第二次从符号位上产生进位了）。

表 12.2　各种数位表示法中的统一操作

	原码	一补码	二补码	负二进制
加法	否	是	是	是
减法	否	是	是	是
乘法	是	否	否	是
除法	是	否	否	否

在执行二进制补码乘法时，如果只需要双字乘积的右半边，那么该操作就是“统一”的，表中相应单元格也要改为“是”。

最后讲讲如何在二进制和负二进制之间转换，以此结束负二进制这一节。

要将二进制转为负二进制，可以先构造一个仅含有正权重位元的字组，然后根据二进制算术中的减法规则，减去另一个仅含有负权重位元的字组。另一个办法更简单：把负权重位置上的位元提取出来，左移一位，然后根据普通二进制算术的减法规则，把提取出来的数从原数中减掉。

要把负二进制转为二进制，可提取奇数位置（也就是权重为 2^n 且 n 为奇数位置）上的位元，左移一位，然后按负二进制加法规则将两数相加。下面举两个例子：

```
     由负二进制转为二进制              由二进制转为负二进制
     1 1 0 1 1 1     (-13)            1 1 0 1 1 1  (55)
   - 1 0 1              二进制减法    +10 1           负二进制加法
   - - - - - -                       - - - - - - -
 … 1 1 1 1 1 0 0 1 1  (-13)          1 0 0 1 0 1 1  (55)
```

在字长大小固定的计算机中，如果把高权重位产生的进位丢弃，那么这些转换方式也适用于负数。为了验证此方法，我们假设字长是6位，并把刚才右上方那个无符号二进制数55视为带符号二进制数−9，然后就能看到转换之后的负二进制数仍然正确。

因为要做负二进制加法，所以在二进制计算机上，无法以软件形式轻松实现出上述负二进制转为二进制的算法。Schroeppel先生[㊀]发明了一种巧妙而实用的方法，能够避开此问题，并且可以执行双向转换［HAK，item128］。要将负二进制转换为二进制，他使用的方法是：

$$\boldsymbol{B} \leftarrow (\boldsymbol{N} \oplus \mathbf{0b10...1010}) - \mathbf{0b10...1010}。$$

为说明其原理，设负二进制数的4个数位分别为$abcd$。然后，故意将其误读为二进制，于是，此数就变成了$8a+4b+2c+d$。异或之后的二进制数是$8(1-a)+4b+2(1-c)+d$。按二进制减法将此数减去$8+2$，可得$-8a+4b-2c+d$，而该数正好是$abcd$的负二进制数。

根据Schroeppel提出的上述公式，很容易就能解出其中的$\boldsymbol{N}$，将它用$\boldsymbol{B}$表示出来，这样就能以3条指令将二进制转为负二进制。将这些技巧总结起来，就可以得出在32位计算机上将负二进制转换为二进制所用的公式：

$$\boldsymbol{B} \leftarrow (\boldsymbol{N} \mathbin{\&} \mathbf{0x5555\ 5555}) - (\boldsymbol{N} \mathbin{\&} \neg\mathbf{0x5555\ 5555}),$$

$$\boldsymbol{B} \leftarrow \boldsymbol{N} - ((\boldsymbol{N} \mathbin{\&} \mathbf{0xAAAA\ AAAA}) \ll \mathbf{1}),$$

$$\boldsymbol{B} \leftarrow (\boldsymbol{N} \oplus \mathbf{0xAAAA\ AAAA}) - \mathbf{0xAAAA\ AAAA}$$

将二进制转换为负二进制所用的公式：

$$\boldsymbol{N} \leftarrow (\boldsymbol{B} + \mathbf{0xAAAA\ AAAA}) \oplus \mathbf{0xAAAA\ AAAA}$$

12.2 以−1+i为底的数制

如果用−1+i作底（其中i为$\sqrt{-1}$），那么所有“复数整数”[㊁]（complex integer，实部和虚部都为整数的复数）都可以表示为一个“数”，无需正负号或其他非常规手段。令人惊讶的是，只需0、1这两种数位，就能表示所有“复数整数”，且每个数只有一种表示方法。本节不打算证明此数制的这一特性或其他属性，只是简要地描述一下。

想要把整数2用−1+i进制表述出来，其实并不那么容易[㊂]。然而可以用算法来转换，把2持续除以底数，并记下余数。这里的“余数”是什么意思呢？除以−1+i之后的余数应尽量化为0或1（这样的话，数位值就只可能是0或1了）。为了演示此想法可

㊀ Richard Schroeppel（1948— ），美国数学家。详情参见：http://en.wikipedia.org/wiki/Richard_Schroeppel。——译者注

㊁ 这种复数在数学上称为“高斯整数”（Gaussian integer），详情参见：http://zh.wikipedia.org/wiki/高斯整数。——译者注

㊂ 感兴趣的读者可以试试。

行，我们任选一个“复数整数” $a+bi$，把它除以 $-1+\mathrm{i}$。现在需要找到满足下式的 q 和 r，其中 q 为“复数整数”，而 r 为 0 或 1：

$$a+bi=(q_r+q_i\mathrm{i})(-1+\mathrm{i})+r$$

q_r 与 q_i 分别表示 q 的实部与虚部。令等式两边的实部与虚部分别相等，并从这两个等式中解出 q：

$$q_r=\frac{b-a+r}{2}，且$$

$$q_i=\frac{-a-b+r}{2}$$

显然，如果 a、b 均为偶数或均为奇数，那么令 r 为 0，即可使 q 成为“复数整数”。此外，若 a、b 一奇一偶，则令 r 为 1，这样就能使 q 为“复数整数”了。

因此，整数 2 可以按照如下方式写为以 $-1+\mathrm{i}$ 为底的数。

由于整数 2 的实部和虚部都是偶数，所以我们知道接下来的除法余数肯定是 0 了，于是直接做除法：

$$\frac{2}{-1+\mathrm{i}}=\frac{2\ (-1-\mathrm{i})}{(-1+\mathrm{i})\ (-1-\mathrm{t})}=-1-\mathrm{t}\text{ 余 }0$$

由于 $-1-\mathrm{i}$ 的实部与虚部均为奇数，所以余数会是 0，于是直接做除法：

$$\frac{-1-\mathrm{i}}{-1+\mathrm{i}}=\frac{(-1-\mathrm{i})\ (-1-\mathrm{i})}{(-1+\mathrm{i})\ (-1-\mathrm{i})}=\mathrm{i}\text{ 余 }0$$

由于 i 的实部与虚部分别为偶数和奇数，所以余数会是 1。对于这种情况，最简单的计算方法就是直接给被除数减 1。

$$\frac{\mathrm{i}-1}{-1+\mathrm{i}}=1(\text{余数是}1)$$

由于 1 的实部与虚部分别为奇数和偶数，所以下一轮除法的余数会是 1。将余数 1 直接从被除数里减去，可得

$$\frac{1-1}{-1+\mathrm{i}}=0(\text{余数是}1)$$

因为商已经是 0 了，所以运算过程结束，从下向上读出余数，可知整数 2 用 $-1+\mathrm{i}$ 进制表示出来是 1100。

表 12.3 列出了 $-1+\mathrm{i}$ 进制的位串 0000 至 1111 以及对应的十进制数，同时也告诉大家：-15 与 $+15$ 之间的整实数如何以 $-1+\mathrm{i}$ 进制来表示。

除了那些与“0”相加的简单规则之外，$-1+\mathrm{i}$ 进制数的加法规则如下：

$$\begin{aligned}
1+1&=1100\\
1+1+1&=1101\\
1+1+1+1&=111010000\\
1+1+1+1+1&=111010001\\
1+1+1+1+1+1&=111011100\\
1+1+1+1+1+1+1&=111011101\\
1+1+1+1+1+1+1+1&=111000000
\end{aligned}$$

表 12.3 在十进制与 −1+i 进制之间转换

n (−1+i 进制)	n 十进制	n 十进制	n (−1+i 进制)	$-n$ (−1+i 进制)
0	0	0	0	0
1	1	1	1	11101
10	$-1+i$	2	1100	11100
11	i	3	1101	10001
100	$-2i$	4	111010000	10000
101	$1-2i$	5	111010001	11001101
110	$-1-i$	6	111011100	11001100
111	$-i$	7	111011101	11000001
1000	$2+2i$	8	111000000	11000000
1001	$3+2i$	9	111000001	11011101
1010	$1+3i$	10	111001100	11011100
1011	$2+3i$	11	111001101	11010001
1100	2	12	100010000	11010000
1101	3	13	100010001	1110100001101
1110	$1+i$	14	100011100	1110100001100
1111	$2+i$	15	100011101	1110100000001

两数相加时，带入某一列的最大进位是 6，所以某一列的最大和就是 8（1 1100 0000）。这使得加法器实现起来相当复杂。如果想构建一台执行复数算术的计算机，那么最好的办法无疑是把实部与虚部分开[⊖]，用二补码等合理方式来表示这两部分的数值。

12.3 以其他数为底的数制

对于以 −1−i 为底的数制，其特性和上节讨论的 −1+i 数制实质上是一样的。如果某个位串在其中一个数制中表示 $a+b$i，那么此位串在另外一种数制中则表示 $a-b$i。

只使用 0、1 这两种数位，也能以 1+i 与 1−i 数制来表示所有“复数整数”。这两个底与另外一对底 −1±i 一样，也互为共轭复数。某些整数如用 1 ± i 进制来表示，则会在左侧出现长度无穷的“全 1 位串”，这与用二补码表示负数类似。正如二补码算术那样，如果要用统一的规则来处理加减法，那么自然就会出现这种情况。比方说整数 2 就是这样，不论用 1+i 进制还是 1−i 进制来表示，都要写成 ...11101100 的形式。因此，这些数制下的加法规则相当复杂，例如 1+1=...11101100。

如果把负二进制中每两个数位编成一对，那么就可以用 −2、−1、0、1 这 4 种数位值来表示四进制正数与负数了。例如：

$$-14_{十进制}=110110_{-2}=(-1)(1)(-2)_4=-1\cdot 4^2+1\cdot -4^1-2\cdot 4^0$$

⊖ 这正是 George Stibitz 于 1940 年在贝尔实验室构建“复数计算器”（Complex Number Calculator）时所用的方法 [Irvine]。

同理，如果在一个 $-1+i$ 进制的“复数整数”中，把每两位编为一组，那么就能用 0、1、-1、$+i$、i 做数位值，以 $-2i$ 进制来表示这个“复数整数”了。这太复杂了些，所以不那么有趣。

“2i 进制”（“quater-imaginary” system，[Knu2]）也类似。这种数制以 2i 为底，用 0、1、2、3 为数位值来表示“复数整数”，这种数制无需正负号。为了表示某些“复数整数”，也就是虚部为奇数的那些数，必须使用小数点右侧一位。例如，以 2i 进制来表示 i，就是 10.2。

12.4　最高效的底是什么

假设要构建一台计算机，现在需决定用何种进制来表示整数。构建寄存器时可使用二态电路（2-state circuit，binary circuit）、三态电路（3-state circuit）、四态电路（4-state circuit）等。那么到底应该选哪个数为底呢？

假设 b 态电路的成本与 b 成正比。也就是说，三态电路的成本比二态电路高 50%，而四态电路的成本是二态电路的两倍，等等。

假设寄存器要能够存放从 0 至某个最大值 M 之间的所有整数。以 b 进制来编码 0 至 M 之间的数需要 $\lceil \log_b(M+1) \rceil$ 个数位（例如，以十进制表示 0 至 999 999 之间的数，需要 $\log_{10}(1\ 000\ 000)=6$ 个数位）。

我们会觉得构建寄存器的成本等于所需数位与每个数位的成本之积：

$$c=k\log_b(M+1)\cdot b$$

其中 c 是寄存器成本，k 为比例常数（constant of proportionality）。对于给定的 M 值来说，要找出令成本最小的 b 值。

如果 b 能令 dc/db 为 0，那么函数就能取到最小值。因此可得

$$\frac{d}{db}(kb\log_b)(M+1)=\frac{d}{db}\left(kb\frac{\ln(M+1)}{\ln b}\right)=k\mathrm{in}(M+1)\frac{\ln b-1}{(\ln b)^2}$$

当 $\ln b=1$，也就是 $b=e$ 时，上式为 0。

这个结果不太理想。因为 $e\approx 2.718$，所以最高效的整数底肯定是 2 或 3。哪一个更高效呢？以 2 为底来构建寄存器所花成本与以 3 为底时的比值是：

$$\frac{c(2)}{c(3)}=\frac{k\cdot 2\log_2(M+1)}{k\cdot 3\log_3(M+1)}=\frac{2\ln(M+1)/(\ln 2)}{3\ln(M+1)/(\ln 3)}=\frac{2\ln 3}{3\ln 2}\approx 1.056$$

因此，以 2 为底的成本高于以 3 为底，不过只多了一点点而已。

以同样方式可分析出：以 2 为底的成本要比以 e 为底时高，约是其 1.062 倍。

12.5　习题

1. 将负二进制转换为二进制所用的 Schroeppel 公式还有一种包含常数 0x5555 5555 的配

套版本，你能找出来吗？

2. 在二进制计算机中，如何通过算术操作与逻辑操作为某个负二进制数加1？例如，0b111⇒0b100。
3. 在二进制计算机中，如何通过算术操作与逻辑操作，把负二进制数沿负方向下调至16的倍数？例如：0b10⇒0b11 0000。
4. 自选一门语言编写程序，把$-1+i$进制下的整数转换成$a+bi$的形式，其中a与b均为实数。例如，当输入程序的值是整数33，或者说0x21时，应该输出5−4i。
5. 如何求$-1+i$进制数的相反数？如何提取$-1+i$进制数的实部？如何提取$-1+i$进制数的虚部？如何求$-1+i$进制数的共轭复数？（$a+bi$的共轭复数是$a-bi$。）

第 13 章
格雷码

13.1 简介

如果每次只能改变 n 个位元中的一个，那么能否由此遍历全部 2^n 种位元组合呢？答案是肯定的，这也正是格雷码（Gray code）的性质。格雷码是一种编码整数的方式，它使得连续两个整数之间只有一个位置上的位元不同。此概念可扩展至以其他数为底的数制上，例如十进制，不过本书只讨论二进制格雷码。

虽说有多种二进制格雷码，不过这里只讲一种，就是“二进制反射格雷码”（reflected binary Gray code）。如果不加限定，那么通常所说的“格雷码”就是指这种。下文经常会不加证明地直接向大家演示如何对表示为格雷码的整数执行基本操作，同时还会展示几个令人惊讶的属性。

二进制反射格雷码的构建方式如下。先从 0，1 开始，这两个位串分别表示整数 0，1：

```
0
1
```

在上述列表底部画一条横轴，然后把整个列表按照此轴向下对折，对折之后的两个值，左侧补 1，对折之前的两个值，左侧补 0：

```
00
01
11
10
```

这就是 $n=2$ 时的二进制反射格雷码了。将上述列表再对折，然后左侧补充 0 或 1，即可得出 $n=3$ 时的格雷码：

```
000
001
011
010
110
111
101
100
```

根据此构建规则，对 n 运用数学归纳法，很容易就能得出：(1) 2^n 种位元组合中的每一种都会出现于列表中，且只出现一次，(2) 只改动1个位元，即可由列表中的某一项推出下一项，(3) 只改动1个位元，即可由列表最末项推出其首项。具备最后一项属性的格雷码叫“循环”(cyclic) 格雷码，二进制反射格雷码一定是循环格雷码。

当 $n>2$ 时，有些格雷码虽然能将 2^n 种值恰好表示一遍，但却不循环 (non-cyclic)。例如 000，001，011，010，110，100，101，111 就是如此。

图13.1列出了在 $n=4$ 时的普通二进制整数和对应的格雷码整数。两组公式演示了如何以逐位的方式 (bit-by-bit level) 在两种表示法之间互化（用硬件实现互相转换时，正需此法）。

那么，能表示 n 位二进制数的格雷码有多少种呢？我们发现：将某种二进制循环格雷码的数值列表在垂直方向上旋转（从 2^n 个位置中的任意一处开始均可，上下环绕着旋转），可生成一套新的格雷码；在水平方向上重新排列各栏，也可生成一套新的格雷码。上述两种操作形式不论是结合起来用还是单独使用，所生成的若干套格雷码都互不相同。所以说，能表示 n 位数的二进制循环格雷码至少有 $2^n \cdot n!$ 套。在 $n \geqslant 3$ 时，这个数字还要大。

由图13.1所列公式可知，格雷码与二进制码之间存在如下双重关系：

- 格雷码整数的第 i 位表示对应二进制整数中，第 i 位及其左侧位元的总体奇偶性（如果第 i 位已经是左界了，则将其左侧位元视为0）。
- 二进制整数的第 i 位表示对应格雷码整数中，从第 i 位到其左界之间所有位元的总体奇偶性。

只需两条指令，即可将二进制码转为格雷码：

$$\boldsymbol{G} \leftarrow \boldsymbol{B} \oplus (\boldsymbol{B} \overset{u}{\gg} \mathbf{1})$$

把格雷码转成二进制码稍难些。一种办法是：

$$\boldsymbol{B} \leftarrow \bigoplus_{i=0}^{n-1} \boldsymbol{G} \overset{u}{\gg} \boldsymbol{i}$$

这个公式曾出现在5.2.1节。正如前面提到的那样，当 $n=32$ 时，也可以按下述代码计算此公式的值。

```
B = G ^ (G >> 1);
B = B ^ (B >> 2);
B = B ^ (B >> 4);
B = B ^ (B >> 8);
B = B ^ (B >> 16);
```

由此可知，该方法一般需要 $2 \cdot \lceil \log_2 n \rceil$ 条指令。

因为从二进制码转为格雷码特别简单，所以很容易就能生成连续格雷码整数：

```
for (i = 0; i < n; i++) {
   G = i ^ (i >> 1);
   output G;
}
```

二进制码	格雷码		
$abcd$	$efgh$		
0000	0000	由二进制码转为格雷码	由格雷码转为二进制码
0001	0001	$e=a$	$a=e$
0010	0011	$f=a\oplus b$	$b=e\oplus f$
0011	0010	$g=b\oplus c$	$c=e\oplus f\oplus g$
0100	0110	$h=c\oplus d$	$d=e\oplus f\oplus g\oplus h$
0101	0111		
0110	0101		
0111	0100		
1000	1100		
1001	1101		
1010	1111		
1011	1110		
1100	1010		
1101	1011		
1110	1001		
1111	1000		

图 13.1　4 位格雷码与二进制码转换公式

13.2　递增格雷码整数

递增 4 位二进制整数 $abcd$ 所需的逻辑可用布尔代数表示如下：

$$
\begin{aligned}
d' &= \bar{d} \\
c' &= c\oplus d \\
b' &= b\oplus cd \\
a' &= a\oplus bcd
\end{aligned}
$$

因此，有一种以硬件构建格雷码计数器的方法，就是先根据上述逻辑构建二进制计数器，然后再如上节图 13.1 所示，依次对 a'、b'、c'、d' 中相邻两个位元取异或，这样就能把输出值转换为格雷码了。

按照下列公式来做可能会稍微好些：

$$
\begin{aligned}
p &= e\oplus f\oplus g\oplus h \\
h' &= h\oplus \bar{p} \\
g' &= g\oplus hp \\
f' &= f\oplus g\bar{h}p \\
e' &= e\oplus f\bar{g}\bar{h}p
\end{aligned}
$$

也就是说，一般规律是

$$G'_n = G_n\oplus(G_{n-1}\bar{G}_{n-2}\ldots\bar{G}_0 p),\quad n\geqslant 2$$

由于表示奇偶性的变量 p 会在 0，1 之间交替，所以计数器电路可以单独用“1 位寄存器”（1-bit register）来维护此值，于每次计数时反转其值即可。

如果想用软件方式找出格雷码整数 G 的下一个数 G'，那么最好的办法也许就是把 G 转为二进制，然后递增此二进制字组，最后再将其转回格雷码。还有一个办法也不错，而且比较有意思，就是设法判断出反转 G 中的哪个位元才能得出 G'。从下面列出的这种数值里选择一个恰当的字组，与 G 取异或，即可得出 G'：

1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16

聪明的读者会发现，上面这些数其实都是掩码，在递增整数 0，1，2，3，… 时，其值发生改变且最靠左的那个位置就是相应掩码中值为 1 的位元所在的位置。因此，要递增格雷码整数，可以先找到 G 所对应的二进制整数，为其加 1，在加法的结果中，其值发生改变且最靠左的那个位置就表示应该反转的那个位元。

上述两种思路可归结为图 13.2，这两段程序都可以递增格雷码整数 G。两种算法都要先用 index（G）把 G 转为二进制。

```
B = index(G);                  B = index(G);
B = B + 1;                     M = -B & (B + 1);
Gp = B ^ (B >> 1);             Gp = G ^ M;
```

图 13.2 递增格雷码整数

用纸笔手算来递增格雷码整数的步骤如下：

从右往左，看当前位元及其左侧位元的整体奇偶性是否为偶。一旦找到了这样的位元，就反转该位置上的值。

或者用下面这种等效的办法：

设 p 为字组 G 的奇偶性。若 p 为偶，则反转其最右侧位元。

若 p 为奇，则找到最靠右且值为 1 的位元，反转其左侧位元。

后一种算法正是上面那个布尔等式所表达的规则。

13.3 负二进制格雷码

如果把整数写为负二进制，然后套用普通二进制转格雷码所用的“移位异或法”，那么就能得到格雷码。3 位格雷码可以表示所有 3 位负二进制数，也就是 −2 至 5 之间的整数。同理，4 位格雷码可以表示所有 4 位负二进制数，也就是 −10 至 5 之间的整数。这种格雷码虽然不是反射格雷码，但也与之相差无几。从 0 和 1 组成的纵向列表开始，先在其上方画一条横轴，把整个列表按此轴对折并补位，然后再于新列表的底部画一条横轴，把整个新列表按此轴对折并补位，以此类推，交替着按上方横轴或下方横轴对折并补位，即可得到 4 位负二进制格雷码。这种格雷码是循环的。

将此种格雷码转回负二进制，其方法当然与把普通二进制反射格雷码转回普通二进

制时所用的方法相同（因为这两种转换操作只是互为逆运算而已，和位串的进制无关）。

13.4 格雷码简史及应用

格雷码得名于弗兰克·格雷（Frank Gray），他是贝尔电话实验室（Bell Telephone Laboratories）的一名物理学家，在 20 世纪 30 年代发明了一种信号传输方法，与当时传输并接收黑白电视信号所用的办法相兼容，后来传播彩色电视信号时用的也是此法，这就是说，如果黑白电视机收到彩色信号，则会以不同程度的灰色来显示图像。

马丁·加德纳㊀在［Gard］一文中讨论了格雷码的应用，其中包括九连环㊁、汉诺塔，以及超立方体图㊂中的哈密顿路径㊃。他还演示了如何将十进制整数转为十进制格雷码。

格雷码也用于位置传感器（position sensor）。条形材质由传导区（conducting area）和非传导区（nonconducting area）构成，二者分别对应格雷码整数的 1 和 0。每一列都摆着一个读取数据用的传导线刷（conducting wire brush）。如果电刷位于两块量化区域之间的分割线上，那么其读数就有两种可能，此时按哪一边来读都行。只会出现一个读数有歧义的传导线刷，无论将其解读为 0 还是 1，都可以对应分界线左侧或右侧的位置。

也可以把条状物做成一系列同心圆轨道，这就是旋转位置传感器（rotational position sensor）。在这种场景下必须使用循环的格雷码才行。图 13.3 演示了这种传感器，其中四个点表示四个电刷。

只用一个包含传导区与非传导区的圆环也可以构建出旋转传感器所用的循环格雷码，然而其代价是要多用一定数量的电刷。这些电刷分布在圆环周围，而不是处于半径上。这种编码叫做单轨格雷码（single track Gray code），或简称 STGC。

现在要找一套格雷码，把它按图 13.1 那样上下排开，要求每一列都必须能由第一列在垂直方向上旋转而得来（由于要在旋转传感器上用，所以还要求必须是循环格雷码）。$n=2$ 时的反射格雷码显然是一套 STGC。下面列出 n 为 2 至 4 时的 STGC。

$n=2$	$n=3$	$n=4$
00	000	0000
01	001	0001

㊀ 马丁·加德纳（Martin Gardner，1914—2010），美国著名的业余数学大师，有很多趣味数学著作，曾为《科学美国人》（Scientific American）杂志的“数学游戏”（Mathematical Games）专栏供稿 20 余年。详情参见：http://zh.wikipedia.org/wiki/马丁·加德纳。——译者注

㊁ Chinese ring puzzle，一种智力玩具，由九个相同的圆环和一把“剑”组成，游戏目的是把九个圆环全套上或卸下。详情参见：http://zh.wikipedia.org/wiki/九连环。——译者注

㊂ 原文为“graphs that represent hypercubes”，应指“Hypercube graph”，在图论中，指超立方体的投影，详情参见：http://en.wikipedia.org/wiki/Hypercube_graph。——译者注

㊃ Hamiltonian path，在图论中，指经过所有节点且只经过一次的路径。详情参见：https://en.wikipedia.org/wiki/Hamiltonian_path。——译者注

11	011	0011
10	111	0111
	110	1111
	100	1110
		1100
		1000

根据 STGC 构建出来的旋转位置传感器更为精简。图 13.4 描述了 $n=3$ 时的旋转 STGC 器件。

这些格雷码都是用非常相似、特别简单而且相当无趣的范式套出来的。根据此模式，$n=5$ 时的那套 STGC 里面有 10 种“代码字”（code word），可以把方向精确到 36 度[⊖]。还可以做得更好。图 13.5 列出了一套 $n=5$ 时的 STGC，其中有 30 种代码字，可以精确到 12 度。与使用 32 种格雷码的最优情况相比，这套方案已经很接近了。

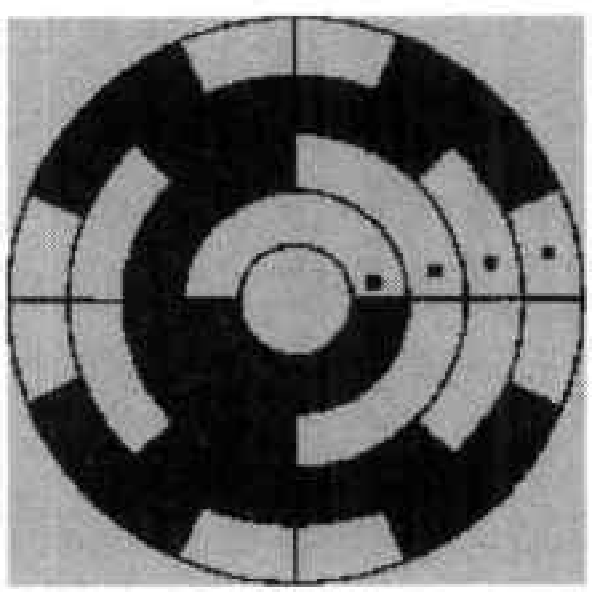


图 13.3 旋转位置传感器

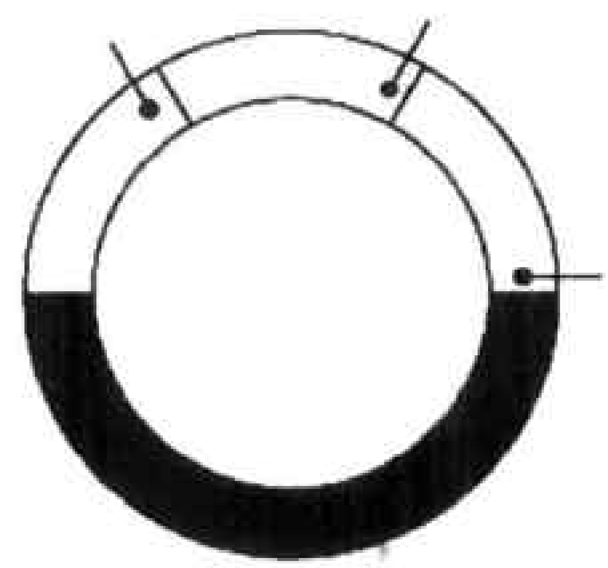


图 13.4 单轨旋转位置传感器

10000	01000	00100	00010	00001
10100	01010	00101	10010	01001
11100	01110	00111	10011	11001
11110	01111	10111	11011	11101
11010	01101	10110	01011	10101
11000	01100	00110	00011	10001

图 13.5 $n=5$ 时的 STGC

本节所列的这些 STGC 都是最佳的，也就是说，当 n 等于 2 至 5 时，对应的那套 STGC 最多可以包含 4、6、8、30 种代码字。

当 $n=9$ 时，可以构建出一套恰好含有 360 个代码字的 STGC（9 是最小值，因为当 $n=8$ 时，无论如何编码，最多只能做出 256 种代码字）[HilPat]。

⊖ 圆周为 360 度，有 10 种不同的读数，所以精确程度是 360 度 ÷10=36 度。——译者注

13.5 习题

1. 请证明：若整数 x 为偶数，则在 x 的二进制反射格雷码 G（x）中，有偶数个值为 1 的位元，若 x 为奇数，则 G（x）中有奇数个值为 1 的位元。
2. “平衡格雷码”（balanced Gray code）是一套循环格雷码，在遍历完其中全部的格雷码后，每一列的位元变动总次数相同。
   (a) 证明：STGC 必须是平衡格雷码。
   (b) 当 $n=3$ 时，能否找到一套具备 8 个代码字的循环格雷码？
3. 编一套表示整数 0 至 9 的循环格雷码。
4. 已知某数的质因子分解式，现在要列出该数的全部因子，要求每个因子都必须由其前一个因子乘以某质因子或除以某质因子而得出。想一个办法来给出此种因子列表。

第 14 章

循环冗余校验

14.1 简介

循环冗余校验（cyclic redundancy check，CRC）是一种数据查错技术，能发现错误，但不能修复。此技术主要用于数据传输。在 CRC 方法中，一定数量的校验位（check bit）会追加至待传输的数据后面，这些校验位合起来也称为校验和（checksum）或哈希码（hash code）。数据在传输过程中可能会出错，而接受者则可决定是否根据这些校验位来核实收到的数据。如果出错，那么接受者就向发送者发出“否定应答”（negative acknowledgment，NAK），请求重发数据。

磁盘驱动器等数据存储设备有时也用此技术。在这种情况下，磁盘中的每一个“块”（block）都有校验位，硬件若侦测到读取错误，则会自动重读该块，或将此错误告知软件。

接下来的文字将站在“发送者”向“接收者”传输“信息”的角度来讲述，不过大家要知道，读写存储设备时也可以用此技术。

14.2 节讲述了 CRC 方法的理论，14.3 节演示了如何用硬件将此理论实现出来，同时还给出一套软件实现方案，也就是大家熟知的 CRC-32 算法。

背景知识

有许多种技术都能生成附加在信息后面的校验位。最简单的办法就是只追加一个位元，它叫做“奇偶校验位”（parity bit），用以表示“码串”（code vector，也就是待传输的信息，奇偶校验位将添加至这段信息后面）里值为 1 的位元个数是奇数还是偶数。如果传输过程中数据里某一位变了，那么新算出来的奇偶校验位要么从偶数变成了奇数，要么从奇数变成了偶数。发送者只需把信息中的所有位元加起来除以 2，然后取余数即可，换句话说，也就相当于对全部位元取异或。接收者把收到的信息中所有位元相加，并与 2 取模，看其结果是否与奇偶校验位相符。也可以说接收者是对包含信息和奇偶位在内的全部位元求和，然后除以 2，看余数是否为 0（假设使用“偶校验位”[⊖]）。

⊖ even parity，在信息中值为 1 的位元个数为偶数时，若使用 0 做奇偶校验位，则称其为“偶校验位”；反之，若使用 1 做奇偶校验位，则称“奇校验位”（odd parity）。——译者注

大家常说这种简单的奇偶校验技术可检测“1 位错误”(1-bit error，信息在传输过程中某一位出错了)。实际上，只要出错的位元(包含奇偶校验位本身)是奇数个，那么就能检测出来，这令人稍感欣慰。比方说，如果信息在传输中有两位出错了，那么此技术无法检测出来，不过要是有 3 位出错了，那么就能检测到了。

对于发送及接收数据的位串行(bit serial)设备来说，生成并检验一个奇偶校验位的所需硬件很容易就能实现出来。只用一个“异或门”(exclusive or gate)再加某种控制电路即可。位并行设备在传输数据时，可使用图 14.1 所示的“异或树”(exclusive or tree)。5.2 节曾讲过如何用软件方式快速算出奇偶校验位。

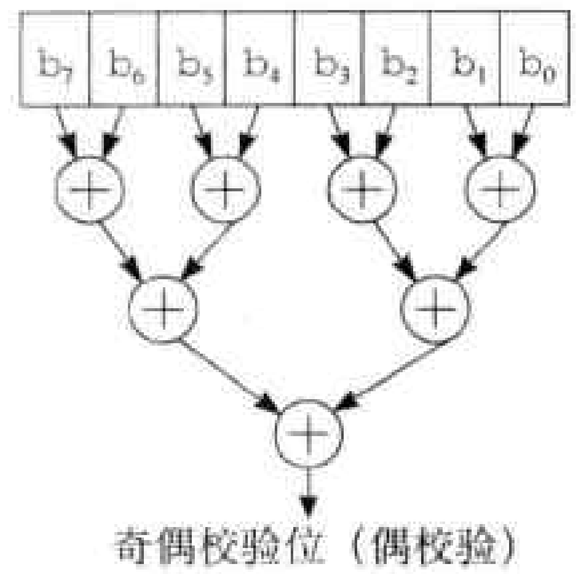


图 14.1 异或树

计算校验和还有其他办法：可以把信息中所有字节取异或，也可以用“首位循环进位”(end-around carry)方式把全部字节加起来。在后面这种方案里，每次求 8 位和时所产生的进位要加到累加器的最低有效位上。与简单的异或法以及丢弃进位的字节求和法相比，此方法据信更容易检测出错误。

有一种技术检测错误的效果据说相当好，而且用硬件实现起来比较容易，这就是循环冗余校验。此技术用另一种方式来计算校验和，这个附加在信息后面的校验和，其长度通常是 8 位、16 位或 32 位。本书将简述其理论，并演示如何用硬件实现，然后给出常用 32 位 CRC 校验和的软件算法。

应该指出的是，想计算数据的校验和或哈希码，还有更为复杂的方式。例如 MD5 和 SHA-1 这两种哈希函数[㊀]分别可算出长度为 128 与 160 位元的哈希码。这些方法主要用于加密，而且与本书要讲的 CRC 法相比，无论用硬件还是软件实现起来都要困难许多。不过，在 Git[㊁]等某些版本控制系统(revision control system)里，SHA-1 只是用来检验数据完整性而已。

14.2 理论

CRC 基于多项式算术(polynomial arithemetic)，具体来说，就是那种将 GF(2)(意思是含有两个元素的伽罗瓦域[㊂])中的两个多项式相除并取余数的那种多项式算术。这就好比是把数据看成一个非常大的二进制数，将其除以另外一个相当大的素数(例如

㊀ 两者详情可分别参见 http://zh.wikipedia.org/wiki/MD5 及 http://zh.wikipedia.org/wiki/SHA 家族。——译者注

㊁ 开源的分布式版本控制系统，由 Linux 核心开发者 Linus Torvalds 创立。详情参见：http://zh.wikipedia.org/wiki/Git。——译者注

㊂ 元素个数有限的域叫伽罗瓦域(Galois field)，也称有限域(finite field)。详情参见：http://en.wikipedia.org/wiki/Finite_field。——译者注

$2^{32}-5$)，并求其余数。此法给出的校验和从直觉上看应该比较可靠。

GF（2）中的多项式就是只包含单变量 x 且系数为 0 或 1 的多项式。系数加减的结果都要和 2 取模，也就是说，此处的加减法均与异或操作等效。例如，将下面两个多项式相加：

$$x^3+x+1 \quad 与$$
$$x^4+x^3+x^2+x$$

其和是 x^4+x^2+1，两式之差也一样。这些多项式系数通常不带负号，不过用了也可以，因为系数值为 -1，与系数值为 1 是等效的。

这种多项式的乘法也比较简单。两个系数之积就等于对二者求“逻辑与”（logical and），然后针对次数相同的项，用异或操作归并其系数。计算 CRC 校验和时不用乘法。

GF（2）中的多项式除法与整数多项式的长除法大致相同。举例如下：

$$
\begin{array}{rllllllll}
 & & & & x^4 & +x^3 & & & +1 \\ \cline{2-9}
x^3+x+1 \big) & x^7 & +x^6 & +x^5 & & & +x^2 & +x & \\
 & x^7 & & +x^5 & +x^4 & & & & \\ \cline{2-5}
 & & x^6 & & +x^4 & & & & \\
 & & x^6 & & +x^4 & +x^3 & & & \\ \cline{3-6}
 & & & & & x^3 & +x^2 & +x & \\
 & & & & & x^3 & & +x & +1 \\ \cline{6-9}
 & & & & & & x^2 & & +1
\end{array}
$$

读者可以验算：将商 x^4+x^3+1 乘以除数 x^3+x+1，再加余数 x^2+1，就等于被除数。

CRC 法将信息看作 GF（2）中的多项式。例如，假设信息是 11001001，传输顺序为从左至右（110...），那么将其看成多项式就是 $x^7+x^6+x^3+1$。传输者与接收者约定了某个固定的多项式，叫做“生成多项式”（generator polynomial）。例如，CCITT（Le Comité Consultatif International Télégraphique et Téléphonique）⊖ 规定的 16 位 CRC 算法，选用 $x^{16}+x^{12}+x^5+1$ 作为生成多项式，此式现已广泛用于计算 16 位 CRC 校验和了。要计算 r 位 CRC 校验和，生成多项式的次数必须是 r。发送者把 m 位信息后面补上 r 个 0，然后将这个 $m+r-1$ 次多项式除以生成多项式。其余数是次数至多为 $r-1$ 的多项式。余数多项式有 r 个系数，于是这些系数就构成了校验和。多项式商丢弃不用。要传输走的数据（“码串”）就是原来那 m 位信息，后面再跟着 r 位校验和。

接收者有两种方式判断所接收数据是否正确。可以根据接收到的前 m 位数据计算校验和，看其是否等于接收到的后面 r 位数据。然而常见的做法则是：接收者用全部 $m+r$ 个位元除以生成多项式，看 r 位余数是否为 0。为了证明在传输无误时余数必为 0，我们将代表信息的多项式记为 M，将传输者算出来的余数多项式记为 R。于是，传输过来的多项式就是 Mx^r-R（也可写成 Mx^r+R）。根据 R 的计算方式可知 $Mx^r=QG+R$，其中

⊖ 现已更名为 ITU-TSS（International Telecommunication Union-Telecommunication Standardization Sector，国际电信联盟-电信标准化部门）。

G 为生成多项式而 Q 为商（商将丢弃）。因此，传输过来的数据 Mx^r-R 就等于 QG，此值显然是 G 的倍数。如果接收者的构建方式与发送者近乎一致，那么它会把收到的数据后面补 r 个 0，再计算余数 R。因为补 0 之后的接收数据仍为 G 的倍数，所以算出来的余数照样是 0。

基本思路就是如此，不过真实的计算过程稍微改良了一下，修正了某些缺陷。比方说，刚才讲的这个办法对传输数据的前导 0 和后缀 0 不敏感，假如传输过程中出错，导致接收到的数据（包括校验和在内）都成 0 了，那么按照上述办法就判断不出错误了。

选一个“合适的”生成多项式也算是门学问，然而不在本书讨论范围内。请记住两个简单的结论：要生成 r 位校验和，G 的次数应为 r，否则校验和的首位总是 0，这样就浪费了此位。同理，最后一个系数应该是 1（也就是说，G 不应被 x 整除），否则校验和的末位总是 0（因为 $Mx^r=QG+R$，假如 G 能被 x 整除，则 R 也必能被 x 整除）。[PeBr] 和/或 [Tanen] 证明了下列关于生成多项式的论述：

- 如果 G 至少有两项，那么一定能检测出“单一位元错误”（single-bit error）。
- 如果 G 不能被 x 整除（也就是说其最后一项是 1），而 e 是使得 G 能够均分 x^e+1 的最小正整数，那么只要在长度为 e 的范围内出现两个错误位元，就一定能检测出来。
- 如果 $x+1$ 是 G 的因子，那么必然能检测到奇数个位元出错的情况。
- r 位 CRC 校验和可检测出所有长度小于等于 r 的“突发错误”（burst error）。（长度为 r 的突发错误是指：在长度为 r 的位串中，首尾两位出错，中间 $r-2$ 位可能正确，也可能有错。）

如果用 $x+1$ 作生成多项式，那么校验和的长度就是 1，这也就相当于给信息追加一个偶校验位（even parity）。（提示：要证明此论述，请思考，当 k 为大于等于 0 的任意值时，x^k 除以 $x+1$ 的余数是什么？）

值得一提的是，如果某种类型的校验码能够检测出所有“两位元错误”和“单一位元错误”，那么从原则上讲，就能据此修正“单一位元错误”。为了理解这一论述，假设接收到的数据中有一位错了。现在依次对每个位元求补，直至遍历完全部位元。那么，除了一种情况外，其余情况都将引发“两位元错误”，而此种错误能检测出来。例外的那种情况就是，对错误的位元求补之后，数据反倒正确了，于是，也就找到了出错的那一位。虽说如此，但是 CRC 算法似乎不是用来修正单一位元错误的。与之相反，接收者在检测到错误后，会要求发送者重新传输整份数据。

14.3 实现

表 14.1 列出了某些常见 CRC 标准所用的生成多项式。其中“Hex”一栏列出了生成

多项式的十六进制写法，由于最高有效位总是1，所以省去不写。

除了生成多项式不同之外，各种CRC标准还有其他区别。大部分标准都有初始化步骤，要把一定数量的非0位元添在待传信息的前面，而其他标准则无此初始化操作。大部分标准都是先传字节中的最低有效位，而某些标准则先传最高有效位。大部分标准都要先追加校验和的最低有效字节，而其他标准则先添加最高有效字节。某些标准会对校验和求补。

CRC-12标准用于传输6位字符流（character stream），其他标准用于传输8位字符，或是以8位元字节为单位的任意数据。IBM的BISYNCH通信标准[1]采用CRC-16。XMODEM[2]、X.25[3]、IBM的SDLC[4]和ISO的HDLC[5]等通信协议都采用CRC-CCITT多项式[Tanen]，也叫ITU-TSS多项式。CRC-32多项式也叫做AUTODIN-II多项式[6]或ITU-TSS多项式（ITU-TSS定义了16位和32位两种多项式）。此多项式用于PKZip[7]、以太网（Ethernet）、AAL5（ATM Adaption Layer 5）[8]、FDDI（Fiber Distributed Data Interface）[9]、IEEE-802 LAN/MAN标准[10]以及某些美国国防部（Department of Defence，简称DOD）的应用场景中。下面要给出的软件算法用的就是CRC-32多项式。

表14.1的前三个多项式都含有 $x+1$ 这个因子，而最后一个多项式（也就是CRC-32）则无此因子。

---

[1] Binary Synchronous Communications（二进制同步通信），也简称BSC或Bisync，详情参见：http://en.wikipedia.org/wiki/Binary_Synchronous_Communications。——译者注

[2] 一种简单文件传输协议，详情参见：http://en.wikipedia.org/wiki/XMODEM。——译者注

[3] 使用电话或者ISDN设备作为网络硬件设备来架构广域网的ITU-T网络协议，详情参见：http://zh.wikipedia.org/wiki/X.25。——译者注

[4] 全称Synchronous Data Link Control（同步数据链路控制），详情参见：http://en.wikipedia.org/wiki/Synchronous_Data_Link_Control。——译者注

[5] 全称High-Level Data Link Control（高级数据链路控制），是一个在同步网上传输数据、面向比特的数据链路层协议，详情参见：http://zh.wikipedia.org/wiki/高级数据链路控制。——译者注

[6] AUTODIN是Automatic Digital Network System（自动数字网络系统）的缩写，也简称ADNS，是美国国防部曾经用过的一种数据通信服务。详情参见：http://en.wikipedia.org/wiki/Automatic_Digital_Network。——译者注

[7] 一种计算机文件压缩程序，引入了流行的ZIP文件格式。详情参见：http://en.wikipedia.org/wiki/PKZIP。——译者注

[8] 一种在异步传输模式（Asynchronous Transfer Mode，简称ATM）网络上传送变长数据包的ATM适配层（ATM adaptation layer），详情参见：http://en.wikipedia.org/wiki/ATM_Adaptation_Layer_5。——译者注

[9] 光线分布式数据接口，由美国国家标准学会（ANSI）制定的一组协议，用于在光缆上发送数字信号。详情参见：https://zh.wikipedia.org/wiki/光纤分布式数据接口。——译者注

[10] IEEE 802是IEEE标准中关于局域网（Local Area Network，LAN）和城域网（Metropolitan Area Network，MAN）的一系列标准。详情参见：https://zh.wikipedia.org/wiki/IEEE_802。——译者注

表 14.1 某些 CRC 码所用的生成多项式

常用名	r	生成器	
		生成多项式	十六进制助记符
CRC-12	12	$x^{12}+x^{11}+x^3+x^2+x+1$	80F
CRC-16	16	$x^{16}+x^{15}+x^2+1$	8005
CRC-CCITT	16	$x^{16}+x^{12}+x^5+1$	1021
CRC-32	32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$	04C11DB7

为了判断收到的信息是否错误地增加或删掉了前导 0，某些协议会把一个或多个非 0 位元放在待传信息的前方。实际上并不传输这些位元，只是用其初始化“CRC 寄存器”(下文会讲到)，计算 CRC 时要用到此寄存器。这些位元通常是 r 个 1。接收者也按相同方式初始化其寄存器。

检查后缀 0 的错误则稍显复杂。接收者可以先根据信息位求出余数，然后与接收到的校验和相比，这样做没问题。然而，接收者要是能把接收到的所有位元（信息及校验和）后面补 r 个 0，然后计算余数，则会更简单些。这样算出来的余数应是 0。如果用余数是否为 0 来判别正误，那么，假设信息后面错误地插入或删除了若干个 0，则计算出来的余数依然是 0，也就是说，根据余数无法检测出此种错误。

此问题的常见做法是，发送者在把校验和添加到信息尾部之前，先对其求补。这样一来，接收者算出的余数一般就不是 0 了，于是，根据余数就能判断出数据尾部是否误增或误删了 0。那么，接收者如何确认信息传输无误呢？

使用“mod”记法来表示余数，可得

$$(Mx^r+R)\bmod G=0$$

用 $\bar{R}$ 来表示多项式 R 求补之后的式子，可得

$$\begin{aligned}(Mx^r+\bar{R})\ \bmod G &= (Mx^r+(x^{r-1}+x^{r-2}+\cdots+1-R))\ \bmod G\\ &= ((Mx^r+R)+x^{r-1}+x^{r-2}+\cdots+1)\ \bmod G\\ &= (x^{r-1}+x^{r-2}+\cdots+1)\ \bmod G\end{aligned}$$

因此，如果信息传输无误，那么接收者算出的校验和应是

$$(x^{r-1}+x^{r-2}+\cdots+1)\ \bmod G$$

只要 G 不变，那么此数就是常量。CRC-32 把下面这个多项式叫做或“余项”（英文写为 residual 或 residue）：

$$x^{31}+x^{30}+x^{26}+x^{25}+x^{24}+x^{18}+x^{15}+x^{14}+x^{12}+$$
$$x^{11}+x^{10}+x^8+x^6+x^5+x^4+x^3+x+1$$

或记为十六进制数 C704DD7B [Black]。

14.3.1 硬件实现

为了用硬件电路计算 CRC 校验和，需要把多项式除法的计算过程化至最简。

硬件算法要引入“移位寄存器”，此处称之为 CRC 寄存器，其长度等于 G 的次数，也就是 r，而不是 $r+1$。在做减法（也就是异或）时，无需表示最高位，因为 G 的最高位和被减数的最高位都是 1。除法过程可用非正规的形式表示如下：

将 CRC 寄存器的所有位元初始化为 0。

　　获取首个/下一个信息位 m。

　　如果 CRC 寄存器的最高位是 1，

　　　　把 CRC 寄存器和 m 视为整体，左移 1 位，然后将结果同 G 的最低 r 位取异或。

　　否则，

　　　　把 CRC 寄存器和 m 左移 1 位。

　　如果信息中还有待处理的位元，则处理下一个。

看上去似乎应该先做减法再移位。假设那样做的话，CRC 寄存器就得存放整个生成多项式了，于是需要 $r+1$ 位才行。反之，如果 CRC 寄存器只包含 G 中权重最低的 r 位，则可先移位，以便对齐操作数。

下面演示了整个计算过程，其中假设 CRC 寄存器所表示的生成多项式 G 是 x^3+x+1，而信息 M 是 $x^7+x^6+x^5+x^2+x$。用二进制表示就是：$G=1011$ 且 $M=1110\ 0110$。

000　初始化 CRC 寄存器。由于高位是 0，所以只需把 0 号信息位向左移入寄存器。

001　因为高位是 0，所以只需把 1 号信息位向左移入寄存器。得出：

011　因为最高位还是 0，所以只需把 2 号信息位向左移入寄存器。得出：

111　因为最高位是 1，所以先移位，再和 011 取异或。得出：

101　因为最高位是 1，所以先移位，再和 011 取异或，得出：

001　因为最高位是 0，所以只需把 5 号信息位向左移入寄存器，得出：

011　因为最高位是 0，所以只需把 6 号信息位向左移入寄存器，得出：

111　因为最高位是 1，所以先移位，再和 011 取异或，得出：

101　因为没有待处理的信息位了，所以寄存器中的值就是余数。

图 14.2 列出了实现这些步骤所用的（简化）电路，也叫做“反馈移位寄存器”(feedback shift register)。3 个方框表示 CRC 寄存器中的 3 位。当信息位进来之后，如果高权重位（就是其中写有 x^2 的那个方框）是 0，则同时将信息位移入 x^0 所在方框，将 x^0 中原有的位移入 x^1，将 x^1 中原有的位移入 x^2，把原有的 x^2 丢弃。如果 CRC 寄存器的高位是 1，那么这个 1 将通过示意图下方的路径进入两个异或门 (exclusive gate)。信息位进来之后，还是按照上述步骤移位，然而由于经过了异或门，所以移位之后，CRC 寄存器里的 3 个位已经和二进制数 011 取过异或了。处理完全部信息位后，CRC 寄存器的值就是 $M \bmod G$。

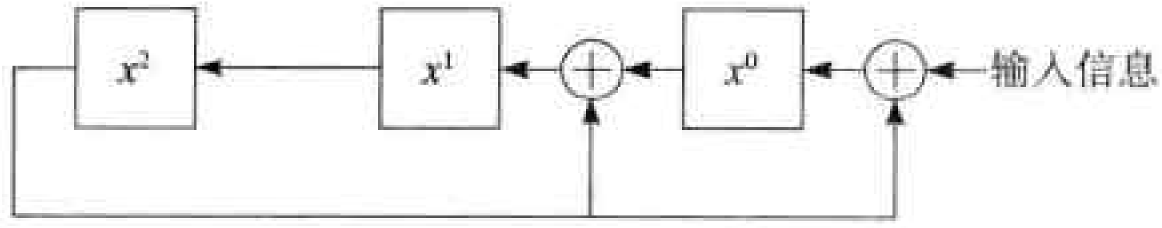


图 14.2　$G=x^3+x+1$ 时所用的多项式除法电路

如果用图 14.2 的电路计算 CRC，那么处理完信息后，必须再给电路填入 r（本例中

为 3）个 0。这样的话，整体处理完之后，CRC 寄存器里面就是想要的校验和，也就是 $Mx^r \bmod G$。只要稍微重排一下电路，就能免去这一步。

按照图 14.3 所示，不从右侧填入信息，而改从左侧填入，这样只需要 r 步。这样做等于把输入的信息 M 预先乘以 x^r。而对多项式来说，“前乘”（premuliplying）和“后乘”（postmultiplying）的效果相同。因此，每个信息位进来之后，CRC 寄存器中的内容就相当于把这部分信息后面添 r 个 0 再求余数。

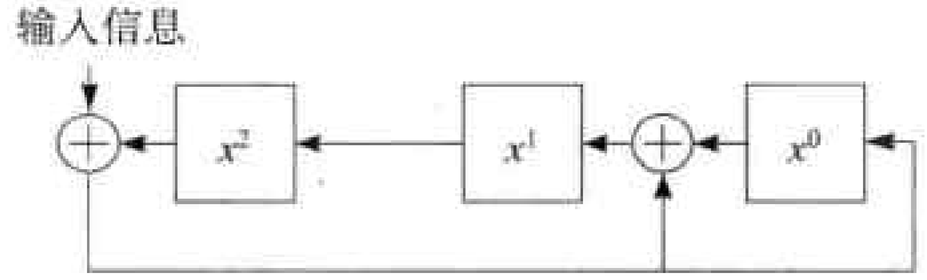


图 14.3 $G=x^3+x+1$ 时所用的 CRC 寄存器电路

图 14.4 画出了适用于 CRC-32 多项式的电路。

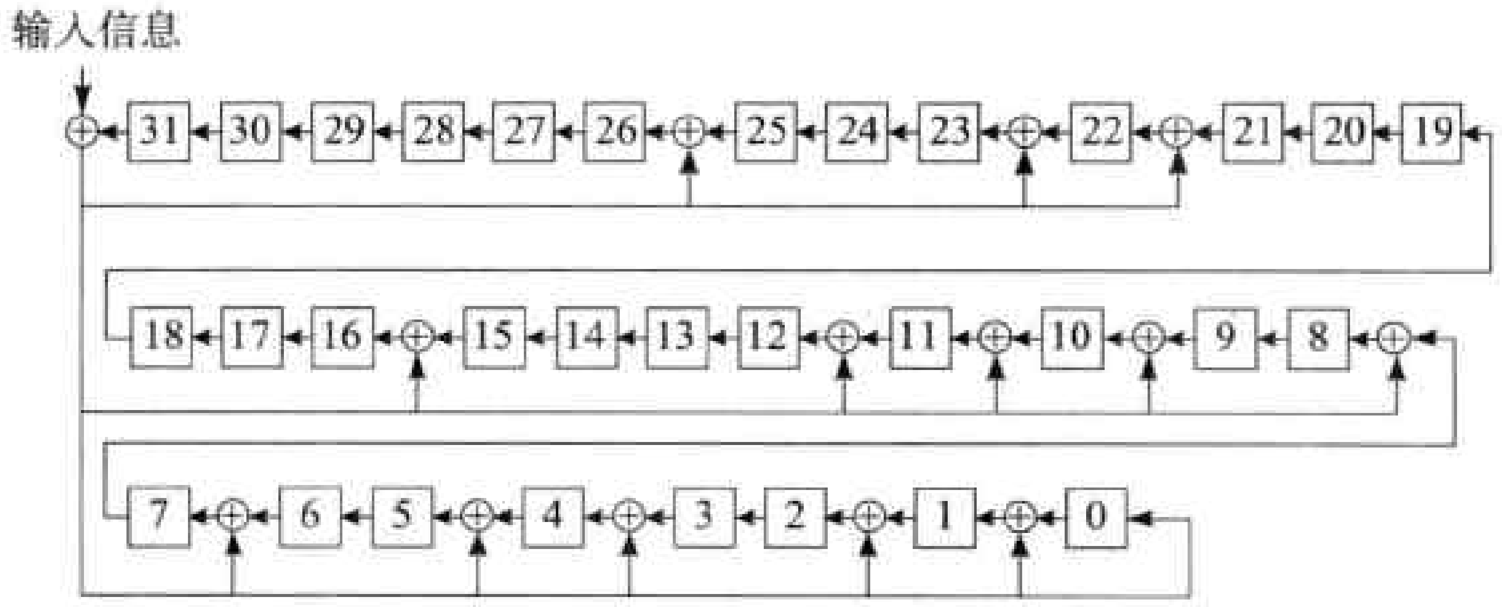


图 14.4 适用于 CRC-32 的 CRC 寄存器电路

14.3.2 软件实现

图 14.5 列出了如何用软件方式实现 CRC-32 的基本算法。CRC32 协议把 CRC 寄存器都初始化为 1，运算时，先传输每个字节的最低有效位，最后对校验和求补。我们假定信息是由整数个字节组成的。

为了尽量与图 14.4 相符，程序用了左移。这需要把每个信息字节水平翻转，并放在 32 位寄存器的左端，程序里用 `byte` 来表示它。为完成此操作，可使用 7.1 节的图 7.1 来执行字组级别位元反转（然而其效率不高，因为此处只需反转 8 位）。

图 14.5 中的代码仅供演示。在每次只处理一个位元的前提下，可极大改善此程序。首先注意到，内层循环的 `if` 语句用完反转过的 `byte` 变量后，就将其丢弃了。而且，`crc` 的高 8 位在内层循环中并未改变（只是移位了而已）。因此，可以在执行内层循环前先执行 `crc=crc^byte`，化简 `if` 语句，并省去循环底部的左移操作。

如果用右移取代左移，那么就能省去两次反转操作。这需要将表示 CRC-32 多项式的十六进制常数反转，并且改为测试 `crc` 的最低有效位。最后，可用某种简单逻辑取代 `if`

测试，以节省分支。修改后的代码列在图 14.6 中。

```
unsigned int crc32(unsigned char *message) {
   int i, j;
   unsigned int byte, crc;

   i = 0;
   crc = 0xFFFFFFFF;
   while (message[i] != 0) {
      byte = message[i];              // Get next byte.
      byte = reverse(byte);           // 32-bit reversal.
      for (j = 0; j <= 7; j++) {      // Do eight times.
         if ((int)(crc ^ byte) < 0)
              crc = (crc << 1) ^ 0x04C11DB7;
         else crc = crc << 1;
         byte = byte << 1;            // Ready next msg bit.
      }
      i = i + 1;
   }
   return reverse(~crc);
}
```

图 14.5 基本 CRC-32 算法

还可以把内层循环展开，将其代码重复 8 次。如果这样修改，那么用图 14.6 来处理输入信息中的单个字节需要执行大约 46 条指令。要执行的指令中，包含一种载入指令及一种分支指令。（有赖编译器将两次 message [i] 加载操作合并，同时把 while 循环变形，只使用一条分支语句，放在变形后的循环底部。）

```
unsigned int crc32(unsigned char *message) {
   int i, j;
   unsigned int byte, crc, mask;

   i = 0;
   crc = 0xFFFFFFFF;
   while (message[i] != 0) {
      byte = message[i];              // Get next byte.
      crc = crc ^ byte;
      for (j = 7; j >= 0; j--) {      // Do eight times.
         mask = -(crc & 1);
         crc = (crc >> 1) ^ (0xEDB88320 & mask);
      }
      i = i + 1;
   }
   return ~crc;
}
```

图 14.6 改进之后的 CRC-32 算法，此算法每次处理一个位元

接下来要讲的这个版本使用查表法。通常计算 CRC-32 时即用此法。上述程序每次处理一个位元，而（通常所用的）查表法每次则能处理一个字节。表格中需要包含 256 个全字常数（fullword constant）。

图 14.6 的内层循环要把寄存器 crc 右移 8 次，每次移位时，如果 crc 最低位是 1，那么还要和常数取异或。这些步骤可以用另一种方式替换：一次性移动 8 位，然后根据 crc 寄存器右 8 位中“1”的模式来构造掩码，把移位后的 crc 同掩码取异或。

我们会发现，计算表格元素的过程和计算单字节 CRC 的过程一样。查表法的代码列在

图 14.7 中。为了令此程序可单独使用，我们把设置表格元素的代码置于其中，第一次使用时，程序会先算好表格。在实际应用中，这些设置表格的步骤可能会放在另外一个函数里，以使 CRC 计算函数尽量简约。还有一种做法是，直接用一长串数组初始化数据（array initialization data）来定义表格。如果用 GCC 编译器把这段程序编译为基本 RISC 指令，那么此函数处理单个输入字节需要执行 13 条指令，其中包含两条加载指令和一条分支指令。

```
unsigned int crc32(unsigned char *message) {
   int i, j;
   unsigned int byte, crc, mask;
   static unsigned int table[256];

   /* Set up the table, if necessary. */

   if (table[1] == 0) {
      for (byte = 0; byte <= 255; byte++) {
         crc = byte;
         for (j = 7; j >= 0; j--) {    // Do eight times.
            mask = -(crc & 1);
            crc = (crc >> 1) ^ (0xEDB88320 & mask);
         }
         table[byte] = crc;
      }
   }

   /* Through with table setup, now calculate the CRC. */

   i = 0;
   crc = 0xFFFFFFFF;
   while ((byte = message[i]) != 0) {
      crc = (crc >> 8) ^ table[(crc ^ byte) & 0xFF];
      i = i + 1;
   }
   return ~crc;
}
```

图 14.7 用查表法实现 CRC 算法

使用常规技巧，还能把上述各版本算法改得再高效一些，不过笔者并未发现可大幅提升性能的修改方法。可以把循环展开，另外，假如编译器没有自动优化加载指令，那么可将其手工编排好。也可以每次从信息串中加载半字或字组（并适当对齐），以便减少载入信息的次数，同时还能减少 crc 与信息取异或的次数（参见习题 1）。如果使用包含 65 536 个字组的表格，那么查表法每次就能处理两个字节了。这样做究竟令程序变得更快还是更慢，还得取决于数据缓存（data cache）的容量以及缓存未命中时的延迟时间。

14.4 习题

1. 请证明：如果生成多项式 G 至少有两项，那么所有“单一位元出错”的情况就必定能检测出来。
2. 修改图 14.7 的主循环，令该程序每次能从信息数据中加载一个字组。为简洁起见，假设信息已经“按全字对齐了”（full-word aligned），而且在末尾没有添加 0 值字节之前，信息长度刚好能用整数个字组表示出来。

第 15 章

纠错码

15.1 简介

本节简单介绍纠错码（error-correcting code，ECC）的理论与实践。我们把讨论范围限定在“二进制向前纠错码块”（binary forward error-correcting block code，“向前纠错”一词简称为 FEC）[㊀] 这个话题上。也就是说，字母表中只有两个符号（本书使用“0”和“1”），如果传输过程中出错，那么接收者可自行修正，既无需向发送者询问更多信息，也不要求其重新发送，所传输的数据由一系列定长的块组成，这些块叫做“代码字”（code word）。

15.2 节讲述了 R. W. Hamming 与 M. J. E. Golay[㊁] 于 1950 年之前各自独立发现的一套纠错码 [Ham]。这是种“单一错误更正”（single error-correction，SEC）技术，Hamming 由此又发现了一套扩展纠错码，叫做“单错误更正双错误检测码”（single error-correcting，double error-detecting，简称 SEC-DED），在检测到两个位元出错时，可更正其中一个位元。

15.4 节回过头来讨论“向前纠错”这一话题，范围仍限于“二进制 FEC 码块”，而此时要解决的基本问题则是：如果给定“块长”（block length，或者说“码长”，code length）、错误检测水平以及纠错能力，那么可以编出多少种不同的代码字呢？

15.2 节主要针对想学习计算机内存纠错码原理的读者。如果您对此话题的相关数学知识感兴趣，或是有志于挑战此领域中尚未解决的数学问题，那么请阅读 15.4 节。

这里要提醒大家，过去 50 年中，ECC 已经发展成为一门博大精深的学问，有大批著作都在讲述此话题及其相关主题，例如 [Hill]、[LC]、[MS]、[Roman] 等，此处只能浅尝辄止，向读者介绍两个重要话题，再谈几个本领域内的术语。纠错码技术中的很多内容极度依赖线性代数（linear algebra）中的记法和结论，同时，也的确是这套抽象理论的一个妙用，然而为了顾及那些对该理论不熟悉的读者，本书避谈此类问题。

本章使用如下记法，后续各节将解释其中的术语。

㊀ “block code”一词又译“分组码”、块码。——译者注

㊁ Marcel J. E. Golay（1902—1989），数学家、物理学家、信息理论家。详情参见：http://en.wikipedia.org/wiki/Marcel_J._E._Golay。——译者注

m “信息”或“消息”中的位元数

k 奇偶校验位的个数（奇偶校验位可简称“校验位”）

n 码长，$n=m+k$

$\boldsymbol{u}$ 信息位串，可展开写为 $\boldsymbol{u}_0$，$\boldsymbol{u}_1$，…，$\boldsymbol{u}_{m-1}$

$\boldsymbol{p}$ 奇偶校验位串，可展开写为 $\boldsymbol{p}_0$，$\boldsymbol{p}_1$，…，$\boldsymbol{p}_{k-1}$

$\boldsymbol{s}$ 错误识别位串（syndrome vector），可展开写为 $\boldsymbol{s}_0$，$\boldsymbol{s}_1$，…，$\boldsymbol{s}_{k-1}$

15.2 汉明码

汉明（Hamming）研发出一套相当直观的纠错码［Ham］，可修正单一位元错误。假设待传输的数据包含一定数量的信息位（记为 $\boldsymbol{u}$），然后给这些位元后面加上若干校验位（记为 $\boldsymbol{p}$），这些校验位可以保证：如果收到的数据中至多有一个位元传错了，那么根据 $\boldsymbol{p}$ 就能判断出那个错误的位元（错误的位元也可以在校验位里面）。具体来说，在汉明码中，如果传输无误，那么 $\boldsymbol{p}$ 就是整数 0，否则，这个整数 $\boldsymbol{p}$ 就表示错误位元所在的位置，该位置从 1 算起。设 m 是信息位的个数，k 是校验位的个数。由于这 k 个校验位在校验信息的同时还必须能校验自身，所以，把 $\boldsymbol{p}$ 视为整数值之后，其范围必须在 0 至 $m+k$ 之间，也就是说，$\boldsymbol{p}$ 可以取 $m+k+1$ 个不同的值。因为 k 个位元可以表示 2^k 种情况，所以必须保证：

$$2^k \geqslant m+k+1 \tag{1}$$

此不等式称为“汉明规则”（Hamming rule）。对于具备单一错误更正能力的二进制 FEC 码块来说，只要待传输的全部位元都必须受检测，那么就肯定符合“汉明规则”。校验位将按照下文描述的方式散布在信息位中。

由于 $\boldsymbol{p}$ 表示错误位（如果有的话）的位置，所以如果奇数位置出错，则 $\boldsymbol{p}$ 的最低有效位必须是 1，若偶数位置出错或无错，则最低有效位必须是 0。有种简单的办法：把 $\boldsymbol{p}$ 的最低有效位记为 $\boldsymbol{p}_0$，令该位是偶校验位（even parity），此位可校验信息块中所有奇数位置上的位元，其本身也放在奇数位置上。接收者检查奇数位置上所有位元（包括 $\boldsymbol{p}_0$ 在内）的奇偶性。若结果是 1，则表明奇数位置出错，若结果为 0，则意味着要么无错，要么偶数位置出错。刚才提的要求是，$\boldsymbol{p}$ 必须指明错误位元的位置，在传输无误时必须为 0，而上述做法符合这一要求。

同理，将 $\boldsymbol{p}$ 的次低有效位（next-from-lease significant bit）记为 $\boldsymbol{p}_1$，令其为偶校验位，它校验 2、3、6、7、10、11、…（二进制的 10、11、110、111、1010、1011、…）等位置，$\boldsymbol{p}_1$ 本身也放在其中某个位置上。在这些位置的二进制表示中，从右往左数，第二个最低有效位是 1。接收者检查这些位置上（也包括 $\boldsymbol{p}_1$ 所在位置）的奇偶性。若结果为 1，则表明这些位置中有一个位置出错，若结果为 0，则要么无错，要么其他位置出错。

继续往下，将 $\boldsymbol{p}$ 的倒数第三个最低有效位记为 $\boldsymbol{p}_2$，此位也是偶校验位，它所校验的那些位置写成二进制，其倒数第三个最低有效位都是 1。也就是说，这些受校验的位置是 4、5、6、7、12、13、14、15、20、…，$\boldsymbol{p}_2$ 本身也放在其中某个位置上。

如果存放校验位的位置都是 2 的幂（1、2、4、8、…），那么其好处是可以各自独立。

也就是说，发送者在计算 p_0 时，无需依赖 p_1、p_2、…，说得更通用些就是：每个校验位均可单独计算，无需依赖其他校验位。

举例来说，现在要编订适用于 $m=4$ 的单一错误更正码。根据（1）式求解 k，令大于等于号变为等号，即可解出 $k=3$。这意味着 k 个校验位的 2^k 种取值全都用上了，所以显得相当高效。具备此种属性的纠错码叫做“完美编码”（perfect code）[⊖]。

本例所说的编码叫做“（7，4）汉明码”，也就是说，码长为 7，信息位的个数是 4。下图演示了校验位 p_i 和信息位 u_i 的位置。

p_0	p_1	u_3	p_2	u_2	u_1	u_0
1	2	3	4	5	6	7

表 15.1 列出了整套编码。这 16 行分别表示 16 种信息位组合所分别对应的汉明校验码。

表 15.1　(7，4) 汉明码

信息	1 p_0	2 p_1	3 u_3	4 p_2	5 u_2	6 u_1	7 u_0
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1
10	1	0	1	1	0	1	0
11	0	1	1	0	0	1	1
12	0	1	1	1	1	0	0
13	1	0	1	0	1	0	1
14	0	0	1	0	1	1	0
15	1	1	1	1	1	1	1

下面演示接收者如何更正单一位元错误，假设代码字

100 1110

已收到。根据表 15.1 第 4 行可知，第 6 位传错了。接收者计算奇数位置上的位元的异或值，得到 0。计算 2、3、6、7 这个四个位置上的位元的异或值，得到 1。最后，计算 4、5、6、7 这四个位置上的位元的异或值，得到 1。于是，错误指示码，或者说错误识别码（syndrome），就是二进制数 110，也等于十进制的 6。接收者只要把第 6 个位元取反，就能修正这块数据了。

⊖ 当 $m=2^k-k-1$ 且 k 为整数时，就能编出完美编码。也就是说，$m=1$、4、11、26、57、120、…。

15.2.1 SEC-DED 码

对于很多应用场景来说，单一错误更正码显得不够理想，因为必须接受所有接收到的码块。SEC-DED 码似乎更安全，而且计算机内存通常使用此种级别的更正与检测技术。

根据 SEC 代码字的全部位元算出一个奇偶校验位（此处假定此校验位能令整个代码字的奇偶性为偶），将其附加在原有的汉明码之中，这就是 SEC-DED 码。这种编码也叫"扩展汉明码"（extended Hamming code）[Hill，MS]。似乎无法直接看出此种编码就是 SEC-DEC 码。为了理解这一点，大家请看表 15.2。预先假设传输过程中会有 0 个、1 个或两个位元出错。根据表 15.2，如果没有错误，那么总体奇偶性（也就是根据接收到的代码字中全部 n 位元算出的奇偶性）就是偶，而根据码块中这（$n-1$）位的 SEC 所算出的错误识别码也是 0。如果有 1 个位元出错，那么接收到的码块其总体奇偶性就是奇。如果出错的那一位就是总体奇偶性校验位，那么算出的错误识别码就是 0。如果错误出在其他位元上，那么错误识别码就不是 0，其值表示发生错误的位置。如果有两个位元出错，那么接收到的码块其总体奇偶性就是偶。若其中一个出错的位元就是总体奇偶性校验位，那么另一个错误必然在码块的 SEC 部分中。此时错误识别码非 0（而且其值就是 SEC 部分中出错的那个位置）。若两个错误均在码块的 SEC 部分里，那么错误识别码也是非 0，不过此时的原因解释起来有点复杂。

表 15.2 为 SEC 添加奇偶校验位，使其变为 SEC－DED 码

可能出现的情况			接收者可由此判断出
出错的位元个数	总体奇偶性	错误识别码	
0	偶	0	无误
1	奇	0 ≠0	整体奇偶性校验位出错 错误识别码表示出错的位置
2	偶	≠0	两个位元出错（无法修正）

原因是，在数个校验位中，必然存在这样一个校验位，它校验了其中一个出错的位置，但却没校验另一个位置。于是，此校验位和它所校验的那些位元其总体奇偶性必是奇数，这就使得错误识别码非 0 了。那么，为什么一定有这种只校验其中一个出错位置而不校验另一个位置的校验位呢？为了理解这一点，首先假设出错的两个位元一个在偶数位置上，另一个在奇数位置上。那么，由于 p_0 这个校验位校验所有奇数位置，而不校验偶数位置，所以奇数位置上的位元其总体奇偶性就是奇，导致错误识别码非 0。说得更通用些，假设两个错误位元的位置是 i 和 $j(i \neq j)$，那么 i 和 j 所对应的二进制数必然在某个位置上有区别，其中一个在该位置上的位元是 1，而另一个是 0。于是，二进制整数中位元值相异的这个位置就对应了一个校验位，把代码字中的位置序号表示成二进制数，只要这个位置上是 1，那么上述校

验位会校验此位置，如果是0，则不校验。这样一来，凡是受此校验位检查的那些位置其总体奇偶性就是奇，这将导致错误识别码非0。例如，假设出错的位置是3和7。用二进制表示这两个位置序号就是0...0011与0...0111。两数在从右至左第3个位置上相异，7所对应的那个位置是1，而3所对应的是0。因此，第3个校验位（也就是校验第4、5、6、7、12、13、14、15、…位的那个校验位）所校验的那些位元其总体奇偶性是奇。

因此，如表格15.2所述，根据整体奇偶性和错误识别码即可判断出错误位元的个数是0个、1个还是两个。在只有一个位元出错时，接收者可修正之。在两个位元出错的情况下，接收者无法判明是只有一个错误位元在SEC部分，还是两个错误位元都在SEC部分。若是前者，则可以修正，若是后者，则不能修正，如果强行修正，就会导致信息位出错。

也可以用偶数位置上的整体奇偶性校验位来取代全体奇偶校验位，因为只要根据前者和奇数位置上的整体奇偶性校验位（也就是校验位里权重最低的那个）就可算出后者。更一般地说，对SEC中某个奇偶校验位所涵盖的位元集合求补集，然后对补集中的所有位元求整体奇偶性，把这个奇偶性位元附加到原SEC后面即可。用硬件实现数据校验时，若使用这一结论，则可节省某些“门”（gate）。

汉明SEC码显然具备最小冗余性（minimum redundancy）。意思是说，对于一定数量的信息位来说，若想更正单一位元错误，就要在其中添加校验位，而汉明SEC码所添加的校验位个数最少。之所以会这样，是因为其构建过程中只包含了必要的校验位，而这些校验位所表示的整数与信息码中的相关位元一一对应，剩下一种情况则表示“传输无错”。换句话说，这套代码符合不等式（1）。给SEC码添加一个全体奇偶校验位，由此构成的SEC-DED码也具备最小冗余性，汉明已经证明了这一论述。他的论证思路是：先假设有一种校验位更少的SEC-DED码，然后得出矛盾，于是据此可推出一开始那种SEC-DED码确实具备最小冗余性。

15.2.2 校验位个数的最小值

表15.3中间一栏列出了m在相关范围内时根据不等式（1）解出的最小k值。右边一栏列出了SEC-DED码所需的最少校验位个数，此数就等于中间那一栏的值再加1。由此表可知，若想给包含64个信息位的内存字组添加SEC-DED级别的纠错码，那么至少需要8个位元，于是，整个内存字组就是72位。

15.2.3 小结

用更偏向数学的ECC术语来说，“汉明码”这个词仅限于上面说的那种“完美编码”，也就是（n，m）为（3，1）、（7，4）、（15，11）、（31，26）等值的纠错码。同理，“扩展汉明码”一词也只用来称呼上述那些“完美SEC-DED纠错码”。然而计算机设计师和工程师所说的“汉明码”则通常指依汉明（Hamming）所述的编码方式构建出来的任何一

套纠错码。他们口中的“扩展汉明码”一词也是如此，任何一种 SEC-DED 码都算扩展汉明码。

表 15.3 修正错误/检测错误所需的额外位元数

信息位的个数 m	SEC 的长度 k	SEC-DED 的长度 k
1	2	3
2 至 4	3	4
5 至 11	4	5
12 至 26	5	6
27 至 57	6	7
58 至 120	7	8
121 至 247	8	9
248 至 502	9	10

首个使用汉明码的 IBM 计算机是 1961 年问世的 IBM Stretch 计算机（型号为 7030）[LC]，它使用（72，64）SEC-DED 码（不是完美编码）。随后还有 1962 年问世的 Harvest 计算机（型号为 7950），它装备有 22 轨磁带驱动器（22-track tape drive），此驱动器使用（22，16）SEC-DED 码。当前计算机所用的纠错码通常不是汉明码，而是根据某种逻辑属性或电器性质编制出来的，这些纠错码可以使奇偶校验树的深度最小，而且还能令其长度均等。汉明码的思路是判断哪个位元有错，而此类纠错码则不同，它们使用硬件查表法。

写作本书时（2012 年），大多数笔记本电脑（notebook personal computer，notebook PC）的内存系统都没有错误检测功能。台式机可能也没有检测机制，或是仅执行简单的奇偶校验。服务器通常使用 SEC-DED 级别的纠错码。

早期“固态计算机”（solid-state computer）中的 ECC 内存通常使用 8 位校验码和 64 个信息位。内存模块（芯片组）一般来说会由 9 个 8 位宽的芯片构成。访问字组（算上校验位，一共 72 位）时，会从这 9 个芯片中取数据，每个芯片取 8 位。根据这些芯片的排布方式，在单个字组中，物理位置相连的 8 个位元会散布于各个芯片中，彼此离得很远。于是，在待访问的字组中，这 72 个位元于物理位置上是分散的。像这样交错排布位元的好处是：如果同一个芯片内的几个相邻位元由于被阿尔法粒子[㊀]或宇宙射线[㊁]击中等原因而改变其值，那么，在物理位置相邻的几个位元里，可能只会有一个位元出错，而这种错误可以更正。某些大型内存采用一种名为 Chipkill 的技术。在整条内存芯片由于断电而

㊀ alpha particle，由两个质子和两个中子组成的放射性粒子。详情参见：http：//zh.wikipedia.org/wiki/α粒子。——译者注

㊁ cosmic ray，是来自外太空的带电高能次原子粒子。详情参见：http：//zh.wikipedia.org/wiki/宇宙线。——译者注

发生故障时，此技术可令计算机继续运转。

交错排布技术也用于通信领域：适时地交错排布位元，可修正突发错误（burst error）。

当今 ECC 内存的结构通常比较复杂，不是只有 8 个检验位和 64 个信息位这么简单。现今的服务器内存也许会用一个 ECC 字组来检测 16 或 32 个信息字节（128 或 256 位）。每个 DRAM[⊖]芯片可能会保存两个、3 个或 4 个物理位置相连的位元。于是，对应的 ECC 就使用 4 个、8 个或 16 个字符做字母表，然而本书在此不讨论这个话题。由于 DRAM 芯片通常是 8 位或 16 位宽，所以内存模块提供的校验位通常比实现 ECC 功能所需的要多。这些额外位元可作其他用途，例如用一两个位元作为内存地址的奇偶校验位。这样的话，内存（或许）就可以判断出它所接收到的地址是不是 CPU 所生成的地址。

与在主内存中使用纠错码相似，现今服务器级别的计算机也许会在高速缓存（cache memory）中使用不同级别的 ECC。此外，ECC 还有可能用于总线（bus）等非内存区域（non-memory area）。

15.3 适用于 32 位信息的软件 SEC-DED 算法

本节要介绍的这种纠错码其编码与解码过程都可用基本 RISC 指令以软件方式高效实现。对 32 位信息来说，可实现单错误更正及双错误检测功能。此技巧与 Hamming 构建汉明码所用思路大致相同。

我们依照 Hamming 的思路构建一些校验位，令接收者很容易就能以软件方式判明出错的位元个数是 0 个、1 个还是两个。与 Hamming 一样，我们也用一个全体奇偶性校验位把 SEC 码转为 SEC-DED 码，同时假定这个全体奇偶性校验位的取值可令其本身与所有受检位的整体奇偶性为偶。由表 15.3 可知，共需 7 个校验位。

首先抛开 DED 属性，只考虑 SEC 问题。构建 SEC 码需要 6 个校验位。用软件实现的主要难点在于，根据 Hamming 所用的构建方法，要把这 6 个校验位同 32 个信息位合并成 38 位值。假设现在需要在 32 位计算机上实现此算法，且信息位保存在 32 位字组中。依照 Hamming 所述方法，发送者要把信息位散布到 38 位量里，然后据此在适当位置上算出校验位，而这一操作实现起来却很棘手，接收者也要面对类似困难。可以把校验位放在一个单独的字组或寄存器中，而把 32 个信息位置于另一个字组或寄存器中。然而如果这样做了，那么每个校验位所校验的那一系列位置就显得不规律了。接下来要描述的这个方案其校验范围比较规律，和 Hamming 的方案差不多（不过 Hamming 所用的方案没有考虑字组边界问题）。受测范围规律了之后，计算起来就容易多了。

表 15.4 列出了每个校验位所校验的范围。该表使用小端序来计算位元位置，0 号位

⊖ Dynamic Random Access Memory，动态随机存取存储器，一种半导体存储器，主要原理是用电容内电荷的多寡来表示二进制位 0 和 1，就是通常所说的计算机"内存"。详情参见：http://zh.wikipedia.org/wiki/动态随机存取存储器。——译者注

表示最低有效位（这与 Hamming 所用的计数方式不同）。

表 15.4 每个校验位所校验的位置

校验位	校验的位置
p_0	0，1，3，5，7，9，11，…，29，31
p_1	0，2 至 3，6 至 7，10 至 11，…，30 至 31
p_2	0，4 至 7，12 至 15，20 至 23，28 至 31
p_3	0，8 至 15，24 至 31
p_4	0，16 至 31
p_5	1 至 31

由此表可知，字组中 32 个位置上的每个信息位都至少有两个校验位来负责校验。比方说，6 号位由 p_1 与 p_2 来校验（此外，p_5 也校验它）。这样的话，如果两个信息字组有一个位元不同，那么所对应的两个代码字（信息位与校验位的合称）之间就会有至少 3 个位元相异（一个是出错的信息位，此外还有至少两个校验位），由此可见，两个代码字之间的“汉明距离”（该词的定义参见 15.4.1 节）至少是 3。此外，当两个信息字中有两个位元值相异的位置时，必然能在 p_0 至 p_5 之间找到至少一个校验位，该位校验了其中一个位置，却没有校验另一个，所以在这种情况下，两个代码字之间的距离也至少是 3。因此，这套方案所制定的纠错码（也就是一套 SEC 码）的最小距离是 3。

假设已经把代码字传给了接收者。把接收到的信息位记为 $\boldsymbol{u}$，校验位记为 $\boldsymbol{p}$，用 $\boldsymbol{s}$（“syndrome”（错误识别码）这个单词的首字母）表示接收者将 $\boldsymbol{p}$ 和 $\boldsymbol{u}$ 取异或后的结果。于是，根据表 15.4 可以推算出当代码字中有 0 个或 1 个位元出错时所对应的错误识别码是多少，表 15.5 列出了这些识别码。

表 15.5 发生 0 个或 1 个错误时，所对应的错误识别码

发生错误的位元	算出来的错误识别码 $s_5 \dots s_0$
（无误）	000000
u_0	011111
u_1	100001
u_2	100010
u_3	100011
u_4	100100
…	…
u_{30}	111110
u_{31}	111111
p_0	000001

（续）

发生错误的位元	算出来的错误识别码 $s_5 \ldots s_0$
$\boldsymbol{p}_1$	000010
$\boldsymbol{p}_2$	000100
$\boldsymbol{p}_3$	001000
$\boldsymbol{p}_4$	010000
$\boldsymbol{p}_5$	100000

举个例子，假设 $\boldsymbol{u}_4$ 这个信息位传错了。查表 15.4 可知，$\boldsymbol{p}_2$ 和 $\boldsymbol{p}_5$ 这两个校验位负责校验 $\boldsymbol{u}_4$。因此，在发送者和接收者各自算出来的校验位中，$\boldsymbol{p}_2$ 与 $\boldsymbol{p}_5$ 这两个位元会与对方不同。在这种情况下，接收者所收到的那份校验位与发送者传出去的相同，于是，错误识别码的 2 号和 5 号位元置位，也就是说错误识别码是 10 0100。

若是校验位里面有一位传错了（而信息位没传错），那么接收者从发送者手里接收到的那组校验位与自己算出来的这组校验位（也就是发送者发出去的那组）相比，除了一个位元不同外，其余位元均相同，而有差别的这个位元就是出错的位元。表 15.5 最后 6 行演示了这种情况。

表 15.5 所列的错误识别码涵盖了代码字传输无误及单一位元误传的全部 39 种情况。因此，错误识别码可以判明数据在传输过程中是否出错，如果有错，还能查到传错的位元所在的位置。此外，如果某个位元传错了，那么无需使用查表法，很容易就能算出误传的位置并更正其位元值。判断逻辑如下：

若 $\boldsymbol{s}=\mathbf{0}$，则无误。
若 $\boldsymbol{s}=\mathbf{011111}$，则 $\boldsymbol{u}_0$ 错误。
若 $\boldsymbol{s}=\mathbf{1xxxxx}$，且 **xxxxx** 非 0，那么 $\boldsymbol{u}$ 中序号为 **xxxxx** 的位元错误。

否则，$\boldsymbol{s}$ 中必有某个位元是 1，这表明某个校验位传错了，只要把错误识别码和收到的校验位（或者是接收者算出来的校验位）取异或，即可将校验位更正。

假设校验位中的错误无须修正，那么就可按下列步骤来执行了，其中 $\boldsymbol{b}$ 是待修正的位元序号。

```
if (s & (s−1)) =0 then...        //无须修正。
else do
        if s=0b01 1111 then b←0
        else b←s &0b01 1111
        u←u ⊕ (1≪b)                  //将 u 中序号为 b 的位元取反。
end
```

有个技巧可以把上述第二个 if-then-else 改写为一条赋值语句。

如果想检测两个位元出错的情况，那么可以计算一个全体奇偶性校验位（校验 $\boldsymbol{u}_{31:0}$

及 $p_{5:0}$之间的所有位元)，把它放在 p 的 6 号位里，一并传出。如果全体奇偶性校验位正确，而错误识别码（$s_{5:0}$）却非 0，那么就可以确认有两个位置传错了。错误识别码非 0 的原因与使用扩展汉明码时相同，15.2.1 节已经解释过了。

图 15.1 与图 15.2 列出了软件实现代码。假定在只有一名发送者与一名接收者的简单情况下，接收者不想更正校验码和全体奇偶性校验位里面发生的错误。

```
unsigned int checkbits(unsigned int u) {

   /* Computes the six parity check bits for the
   "information" bits given in the 32-bit word u. The
   check bits are p[5:0]. On sending, an overall parity
   bit will be prepended to p (by another process).

   Bit   Checks these bits of u
   p[0]  0, 1, 3, 5, ..., 31 (0 and the odd positions).
   p[1]  0, 2-3, 6-7, ..., 30-31 (0 and positions xxx1x).
   p[2]  0, 4-7, 12-15, 20-23, 28-31 (0 and posns xx1xx).
   p[3]  0, 8-15, 24-31 (0 and positions x1xxx).
   p[4]  0, 16-31 (0 and positions 1xxxx).
   p[5]  1-31 */

   unsigned int p0, p1, p2, p3, p4, p5, p6, p;
   unsigned int t1, t2, t3;

   // First calculate p[5:0] ignoring u[0].
   p0 = u ^ (u >> 2);
   p0 = p0 ^ (p0 >> 4);
   p0 = p0 ^ (p0 >> 8);
   p0 = p0 ^ (p0 >> 16);        // p0 is in posn 1.

   t1 = u ^ (u >> 1);
   p1 = t1 ^ (t1 >> 4);
   p1 = p1 ^ (p1 >> 8);
   p1 = p1 ^ (p1 >> 16);        // p1 is in posn 2.

   t2 = t1 ^ (t1 >> 2);
   p2 = t2 ^ (t2 >> 8);
   p2 = p2 ^ (p2 >> 16);        // p2 is in posn 4.

   t3 = t2 ^ (t2 >> 4);
   p3 = t3 ^ (t3 >> 16);        // p3 is in posn 8.

   p4 = t3 ^ (t3 >> 8);         // p4 is in posn 16.

   p5 = p4 ^ (p4 >> 16);        // p5 is in posn 0.

   p = ((p0>>1) & 1) | ((p1>>1) & 2) | ((p2>>2) & 4) |
       ((p3>>5) & 8) | ((p4>>12) & 16) | ((p5 & 1) << 5);

   p = p ^ (-(u & 1) & 0x3F);   // Now account for u[0].
   return p;
}
```

图 15.1 计算校验位

计算校验位时，checkbits 函数先跳过信息位 u_0，然后按如下式子计算 $0 \leqslant i \leqslant 4$ 时的校验位 k_i，计算每个校验位时，要跳过序号为 i 的那一行：

$$0. \qquad (x \leftarrow u \oplus (u \overset{u}{\gg} 1))$$

$$1. \qquad (x \leftarrow u \oplus (u \overset{u}{\gg} 2))$$

2. $\left(x \leftarrow u \oplus \left(u \overset{u}{\gg} 4\right)\right)$

3. $\left(x \leftarrow u \oplus \left(u \overset{u}{\gg} 8\right)\right)$

4. $\left(x \leftarrow u \oplus \left(u \overset{u}{\gg} 16\right)\right)$

```
int correct(unsigned int pr, unsigned int *ur) {

   /* This function looks at the received seven check
   bits and 32 information bits (pr and ur) and
   determines how many errors occurred (under the
   presumption that it must be 0, 1, or 2). It returns
   with 0, 1, or 2, meaning that no errors, one error, or
   two errors occurred. It corrects the information word
   received (ur) if there was one error in it. */

   unsigned int po, p, syn, b;

   po = parity(pr ^ *ur);        // Compute overall parity
                                 // of the received data.
   p = checkbits(*ur);           // Calculate check bits
                                 // for the received info.
   syn = p ^ (pr & 0x3F);        // Syndrome (exclusive of
                                 // overall parity bit).
   if (po == 0) {
      if (syn == 0) return 0;    // If no errors, return 0.
      else return 2;             // Two errors, return 2.
   }
                                 // One error occurred.
   if (((syn - 1) & syn) == 0)   // If syn has zero or one
      return 1;                  // bits set, then the
                                 // error is in the check
                                 // bits or the overall
                                 // parity bit (no
                                 // correction required).

   // One error, and syn bits 5:0 tell where it is in ur.

   b = syn - 31 - (syn >> 5); // Map syn to range 0 to 31.
// if (syn == 0x1f) b = 0;    // (These two lines equiv.
// else b = syn & 0x1f;       // to the one line above.)
   *ur = *ur ^ (1 << b);      // Correct the bit.
   return 1;
}
```

图 15.2 接收者执行的校验操作

这样会令每个 p_i 位都能校验字组 x 中不同位置上的位元，如图 15.1 所示。计算 p_5 时，上面 5 条赋值语句都用上了。计算每个校验位时所用的一组赋值语句都很规律，这样做的好处是许多相似代码都可以合并起来。从图 15.1 来看，原本需要条 4×5+5=25 条赋值语句，而优化之后则可缩减至 15 条。

顺便说句，若是计算机有指令能算出字组的奇偶性，或者有种群计数指令（调用该指令后，目标寄存器的最低有效位就表示待测字组的奇偶性），那么就无需按照上面那种规律的模式来计算校验位了。在此种计算机上，可按如下方式计算：

```
p0 = pop(u & 0xAAAAAAAB) & 1;
p1 = pop(u & 0xCCCCCCCD) & 1;
```

以此类推。

把 6 个校验位都打包进 p 这一个变量之后，checkbits 函数回过头来处理 u_0，如果 $u_0=1$，那么就把这 6 个校验位都取反。（可参阅表 15.4，由于代码在计算 p_5 时已经把 u_0 错误地包含在内了，所以后面如果发现 $u_0=1$，就必须把 p_5 取反。）

15.4 广义错误修正

本节继续谈二进制 FEC 码块，不过讲得要比 15.2 节所述汉明码更通用一点。原来的假设是码块必须由信息位和校验位两个独立集合构成，而且代码字的个数必须是 2 的幂，本节抛掉这些假设。此外，还要研究一些纠错、查错能力比 SEC 和 SEC-DED 要高的方案。比方说，需要一种双错误更正码，令其适用于以二进制形式表示的十进制数。如果一套编码中有 16 种代码字（其中 10 种用于表示十进制数位，另外 6 种未使用），那么每个代码字至少要有 11 个位元。然而若代码字只有 10 种，那么每个代码字就可以做到 10 位长了。（在 15.4.3 节的表 15.8 中，$d=5$ 这一栏描述了此论断，下文会解释其原因。）

一套“编码”（code）就是由若干代码字（code word）所组成的集合，本书假定所有代码均为二进制位串，且长度相等，前面讲过，这个长度就叫“码长”（code length）。代码字所表示的具体意义不限，比方说，可以表示字母与数字[⊖]，也可表示图片中的像素值。

举个简单的例子，下面这套编码与整数 0 至 7 的二进制写法相对应，其中每个位元都重复了 3 次：

{0 0000 0000，0 0000 0111，0 0011 1000，0 0011 1111，1 1100 0000，…，1 1111 1111}

再举个例子，“五中取二码”（two-out-of-five code）的每个代码字里都有两个位元是 1：

{00110，00011，00101，01001，01010，01100，10001，10010，10100，11000}。

这套编码有 10 种码字，因此适合表示十进制数位。请注意，如果用代码字 00100 表示十进制 0，那么剩下 9 个值就可以分别表示 1 至 9 这 9 个十进制数位了，在这 9 个代码字中，从左至右，各个位元的权重分别是 6，3，2，1，0。

“码率”（code rate）是用来衡量编码效率的一种手段。对于汉明码这类编码来说，可将码率定义为信息位的个数除以码长。前文提到的那种汉明码，其码率为 $4/7\approx0.57$。更为通用的定义则是：以 2 为底，把代码字的种数取对数，然后除以码长。刚才提到的那两套简单编码，其码率分别为 $\log_2(8)/9\approx0.33$ 与 $\log_2(10)/5\approx0.66$。

⊖ alphanumeric，也称“文数字”，是拉丁字母与数字字符的合称。详情参见：http：//zh.wikipedia.org/wiki/文数字。——译者注

15.4.1 汉明距离

ECC 理论的核心概念就是汉明距离（Hamming distance）。两个（等长）代码字的汉明距离就是其二进制值相异的位置个数。换一种说法，汉明距离是两个代码字取异或之后的种群计数。不妨将其视为一种“距离函数”（distance function），因为其符合线性代数中对距离函数所下的定义：

$$
\begin{aligned}
&d(\mathbf{x},\mathbf{y})=d(\mathbf{y},\mathbf{x}),\\
&d(\mathbf{x},\mathbf{y})\geqslant 0,\\
&\text{当且仅当 } \mathbf{x}=\mathbf{y} \text{ 时，} d(\mathbf{x},\mathbf{y})=0,\\
&d(\mathbf{x},\mathbf{y})+d(\mathbf{y},\mathbf{z})\geqslant d(\mathbf{x},\mathbf{z}) \qquad \text{（三角不等式）}
\end{aligned}
$$

本书用 $d(\mathbf{x},\mathbf{y})$ 表示代码字 $\mathbf{x}$ 与代码字 $\mathbf{y}$ 之间的汉明距离，为简洁起见，也称为 $\mathbf{x}$ 与 $\mathbf{y}$ 的距离。

假设某套编码的最小距离是 1。现在从这套编码中挑出两个码字 $\mathbf{x},\mathbf{y}$，二者只有一个位置上的位元不同。如果 $\mathbf{x}$ 中和 $\mathbf{y}$ 相异的那个位元在传输过程中出错了，那么其值就会翻转，这样一来，就有两种可能，要么 $\mathbf{x}$ 中的那个位元传错了，要么发送者所发的本来就是 $\mathbf{y}$，而这个 $\mathbf{y}$ 在传输过程中没有出错。接收者显然无法判断出到底是哪一种情况。因此，这样一套编码通常连单个位元出错的情况都检测不出来。

假设某套编码的最小距离是 2。如果只传错了一位，那么接收者会接收到错误的代码字，于是（原则上）可以侦测出错误。如果传错了两位，那么可能恰好与另外一个有效的代码字重合。于是，接收者就无法侦测双位元错误了。此外，也无法更正单一位元错误。假如接收者收到的代码字中有一个位置传错了，那么便无法推断出原本正确的代码字，因为在有效的那些代码字里可能会有两种代码字，其传错了一个位元之后的值恰好和接收者现在收到的值一样。

给信息位后面再加一个奇偶校验位，即可构成刚才所说的那种编码。下面列出了一套适用于 3 个信息位（也就是当 $m=3$ 时）的代码字。最右边那一位是奇偶校验位，该值令 4 个位元的整体奇偶性为偶。读者可以验证，两代码字之间的最小距离是 2。

0000
0011
0101
0110
1001
1010
1100
1111

实际上，增加一个奇偶校验位之后，就可以检测出奇数个位元出错的情况，然而通常说一套编码能检测出 k 个位元错误，意思却是指多达 k 个位元出错的每一种情况都能检测出来[⊖]。

现在思考两个代码字最小距离为 3 的情况。如果传输过程中有 1 位或两位出错，那么接收者会收到无效代码字。若是仅有 1 位传错，那么可以想见，接收者会依次翻转每一个位元，看看翻转哪个位元后能够得到有效的代码字。于是，这套编码可令接收者检测并更正单一位元错误。如果两个位元出错，那么有两种情况：要么是某个有效代码字在传输过程中真的有两个位元都传错了，要么是另外一个有效的代码字在传输过程中有一个位元出错，而接收者无法分辨出到底是哪种情况，所以说，无法修正双位元错误。

同理可推出：两代码字最小距离为 4 的一套编码能够令接收者更正所有单一位元错误，并侦测出双位元错误（这是一套 SEC-DED 码）。正如本章前文提到的那样，计算机内存通常使用具备此种查错与纠错能力的编码。

表 15.6 列出了码块在各种最小距离下所具备的纠错与查错能力。

可以牺牲一些纠错能力，以换取查错能力。例如现在要编一套最小距离为 3 的编码，那么在这种冗余程度下，我们可以放弃纠错能力，而令其能侦测出单一位元错误及双重位元错误。在最小距离为 5 的情况下，可编一套能够更正单个位元错误并侦测 3 个位元错误的编码，或是一套不具更正能力但却能侦测出 4 个位元错误的代码，以此类推。在表 15.6 中，可以从“纠错能力”这一栏里减去一个数，将其加到“查错能力”这一栏中。

表 15.6 编码在各种最小距离下所能更正/检测的错误位元个数

最小距离	纠错能力（以位元计）	查错能力（以位元计）
1	0	0
2	0	1
3	1	1
4	1	2
5	2	2
6	2	3
7	3	3
8	3	4
d	$\lfloor (d-1)/2 \rfloor$	$\lfloor d/2 \rfloor$

15.4.2 编码论的主要问题

我们已经思考过的问题是：“在已知信息位个数 m 及最小距离 d 的情况下，需要多少个校验位？”为了讲得更加通用些，现在反过来思考：“在已知码长 n 及最小距离 d 的情

⊖ 意思是，这套编码要能令接收者侦测出：传输无误、有 1 个位置出错、有两个位置出错、……、有 $k-1$ 个位置出错、有 k 个位置出错等所有情况。上述代码虽然能检测出 3 个位元传错的情况，但是却没法检测到两个位元传错的情况，所以不能称为“3 位元错误检测码”。——译者注

况下，所制定出的编码里能包含多少种代码字？”这样一来，代码字的种数就未必需要是2的整数幂了。

此处参考［Roman］等教材，设码长为 n 且最小距离为 d 的（二进制）编码最多能表示 $A(n,d)$ 种代码字。本节其余部分将探讨有关此函数的某些研究成果。该函数的求值问题又称“编码论的主要问题”（the main coding theory problem）［Hill，Roman］。本节假设 $n \geqslant d \geqslant 1$。

基本上很容易看出：

$$A(n,\ 1)=2^n \tag{2}$$

这是因为在码长为 n 的情况下，有 2^n 种不同的取值。

在最小距离为2的情况下，从前面所举的具备一个奇偶性校验位的那套编码来推断，$A(n,2) \geqslant 2^{n-1}$。但是，$A(n,2)$ 不能超过 2^{n-1}，原因如下。假设有一套码长为 n 且最小距离为2的编码，其可以表示的代码字种数大于 2^{n-1}。从整套代码字中随意删掉一列。（假设代码字按照15.2节中表15.1那样排成矩阵。）现在这套代码字的码长为 $n-1$，而最小距离至少是1（删掉某列后，至多可令最小距离减1），但是，其代码字的种数却超过了 2^{n-1}。也就是说，$A(n-1,1)>2^{n-1}$，这与等式（2）矛盾。因此，

$$A(n,2)=2^{n-1}$$

目前看来问题还不算难。可是 $A(n,3)$ 呢？此问题从某种意义上说还未解开，因为不能用公式或简单的办法求出其值。当然了，可以求出 $A(n,3)$ 函数在一些特殊情况下的值，而且还能知道某些取值范围，不过大多数情况下，无法确知其值。

（1）式在取等号时，表示该问题在 $d=3$ 时的解。设 $n=m+k$，则（1）式可改写为

$$2^m \leqslant \frac{2^n}{n+1} \tag{3}$$

此处 m 是信息位的个数，所以 2^m 就是代码字的最大种数。因此可知

$$A\ (n,3) \leqslant \frac{2^n}{n+1}$$

当 $2^n/(n+1)$ 为整数时，上式可取等号（这正是 Hanmming 构建编码时的情况）。

例如，当 $n=7$ 时，有 $A(7,3)=16$，这也就是15.2节说的（7，4）汉明码。当 $n=3$ 时，有 $A(3,3) \leqslant 2$，含有000与111两个代码字的这套编码印证了不等式取至上限2时的情况。当 $n=4$ 时，有 $A(4,3) \leqslant 3.2$。在草稿纸上演算一下就知道，此时无法制订出一套码长为4且 $d=3$ 的编码。因此，如果（3）式无法取等号，那么给出的仅仅是理论上限值而已，实际上未必能制订出一套含有那么多种代码字的编码来。

当 $n \geqslant 2$ 时，有个比较有趣的关系式：

$$A(n,d) \leqslant 2A(n-1,d) \tag{4}$$

因此，在保持最小距离 d 不变的前提下给码长加1，至多能令代码字的种数翻倍。为理解其中道理，假设码长为 n，距离为 d，代码字种数为 $A(n,d)$。在整套编码的代码字中任选一栏。这样一来，可能会有一半或一半以上的代码字在该栏中的位元为0，也可能会有

一半或一半以上的代码字在该栏中的位元为 1——这两个论述中至少有一个成立。在以“0”、“1”为标准所划分的两个子集中，选择至少包含 $A(n,d)/2$ 种代码字的那一个，将该列从这个子集中删去，于是就有了一套新的编码。这套编码的 n 值比原来少 1，d 不变，而且至少有 $A(n,d)/2$ 种代码字。因此 $A(n-1,d)\geqslant A(n,d)/2$，这与不等式（4）相符。

再讲一个比较有用的关系式。当 d 为偶数时，

$$A(n,d)=A(n-1,d-1) \tag{5}$$

为理解其道理，假设有套码长为 n 且最小距离为奇数 d 的编码，将其记为 C。为其中每个代码字添加一个奇偶校验位，令该代码字的整体奇偶性为偶，这样就有了一套新编码，其码长为 $n+1$，而代码字的个数与 C 相同，其最小距离是 $d+1$。因为假如 C 中两个代码字间距是 x，而 x 为奇数，那么其中一个代码字的奇偶性必然是偶，而另一个的奇偶性则必然是奇。对于前者，所添的奇偶校验位是 0，而后者则是 1，这会把两个代码字之间的距离扩大为 $x+1$。若两个代码字的距离 x 为偶数，则给两个代码字都添 0，于是，其间距不变。因为 d 是奇数，所以原来间距为 d 的任意两个代码字现在的间距就变为 $d+1$。原来间距大于 d 的两个代码字，现在的间距要么不变，要么增加。因此，这套新编码的最小间距就是 $d+1$。这说明在 d 为奇数时，$A(n+1,d+1)\geqslant A(n,d)$，或者说，在 d 为大于等于 2 的偶数时，$A(n,d)\geqslant A(n-1,d-1)$。

现在假设有一套码长为 n 且最小距离 d 大于等于 2（奇偶不限）的编码。随意删掉一栏，就可得出一套新编码。其长度为 $n-1$，而最小间距至少是 $d-1$，而且其代码字的种数与原来那套相同（因为新代码的最小距离至少是 1，所以其中的每个代码字都互不相同）。因此 $A(n-1,d-1)\geqslant A(n,d)$。与刚才的不等式合起来，就变为等式（5）了。

15.4.3 *n* 维球面

在 d 为大于等于 1 的任意值时，从 n 维球面（n-dimensional sphere）的角度思考，可以推出 $A(n,d)$ 的上下界。给定某个代码字，可以假想一个半径为 r 的“球体”，将该代码字置于球体中心，与其汉明距离小于等于 r 的全部代码字都在这个球里。

在这个半径为 r 的球体里，有多少个点（代码字）呢？首先思考，在球面上有多少个与球心代码字的汉明距离恰好为 r 的点？从 n 个物体中选 r 个，不计顺序，这种组合的个数就是刚才那个问题的答案。从位于中心点的代码字里选取 r 个位元，翻转其值，这样得出来的代码字与原代码字的汉明距离恰好为 r。这个“选取”函数经常写为 $\binom{n}{r}$，其值可由下式算出[⊖]：

$$\binom{n}{r}\frac{n!}{r!\ (n-r)!}$$

⊖ 该值也称“二项式系数”（binomial coefficient），因为把二项式 $(x+y)^n$ 展开后，$x^r y^{n-r}$ 这一项的系数正是 $\binom{n}{r}$。

因此，$\binom{n}{0}=1$，$\binom{n}{1}=n$，$\binom{n}{2}=\frac{n(n-1)}{2}$，$\binom{n}{3}=\frac{n(n-1)(n-2)}{6}$，依此类推。

在半径为 0 至 r 的各个球面上，统计点的个数，将其加总，于是得出半径为 r 的球体内所含点的总数：

$$\sum_{i=0}^{r}\binom{n}{i}$$

似乎找不到简单的求和公式［Knu1］。

由此很容易求出 $A(n,d)$ 的取值边界。首先，假设有一套码长为 n 最小距离为 d 且包含 M 种代码字的编码。把每个编码都放在一个球体中心，然后在不使任意两球有公共点且每个球体半径都相等的情况下，尽力扩大其半径。若 d 为奇数，则半径是（$d-1$）/2，若 d 为偶数，则半径是（$d-2$）/2（参见图 15.3）。由于每个点至多包含在一个球体内，所以 M 个球体内的总点数必须小于等于此空间内的总点数。也就是说：

$$M\sum_{i=0}^{\lfloor (d-1)/2\rfloor}\binom{n}{i}\leqslant 2^n$$

上式对任意 M 值成立，因此，当 $M=A(n,d)$ 时，有

$$A(n,d)\leqslant\frac{2^n}{\sum_{i=0}^{\lfloor (d-1)/2\rfloor}\binom{n}{i}}$$

这叫做“球体装填上限”[⊖]（sphere-packing bound）或“汉明界”（Hamming bound）。

用球体这个思路来推算 $A(n,d)$ 的下限也很容易。还是假设有一套编码，其码长为 n，最小距离为 d，且包含了尽可能多个代码字，也就是 $A(n,d)$ 种代码字。将每个代码字均置于半径为 $d-1$ 的球体中心。于是，空间内 2^n 个点肯定全都囊括在这些球体中（球体之间可相交）。如果不是这样的话，那么必然存在一个点，其与所有代码字的距离都大于等于 d，而这不可能，因为这等于说该点也应该成为代码字，因而此论述与假设矛盾。于是，就得到弱形式的 Gilbert-Varshamov 界（Gilbert-Varshamov bound）：

$$A(n,d)\sum_{i=0}^{d-1}\binom{n}{i}\geqslant 2^n$$

对线性码（linear code）来说，也有强形式的 G-V 界。其推导需要用到线性代数的方法，这些方法对于线性码这一主题很重要，然而本书只打算简介一下纠错码，所以不包含这部分内容。可以这样说，如果某套编码中任意两个代码字之和（也就是两者的异或值）也是代码字，那么这套编码就是线性码。表 15.1 所列汉明码就是线性码。由于 G-V界是线性码的下界，所以对本节所讨论的这种不加限制的编码来说，自然也是其下界了。在 n 较大时，该式给出的下界对线性码和不受限的编码来说，都是已知

⊖ 又称“最密堆积”，详情参见：https://zh.wikipedia.org/wiki/最密堆积。——译者注

方案里最好的。

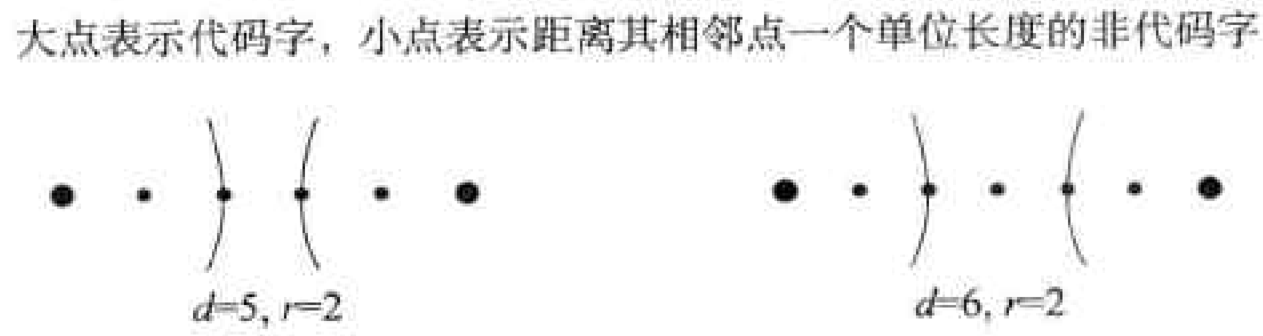


图 15.3 球体最大半径，该取值能尽量把可更正的点囊括于球体内

强 G-V 界给出的限定范围是：$A(n,d) \geqslant 2^m$，其中 m 是令下式成立的最大整数：

$$2^m < \frac{2^n}{\sum_{i=0}^{d-2} \binom{n-1}{i}}$$

也就是说，把不等式右侧“严格”下调[⊖]为距其最近的 2 的幂，令 2^m 与之相等，此时 m 最大。对于 $(n,d)=(8,3)$、$(16,3)$ 以及退化为 $(6,7)$ 的情况来说，一定要“严格”下调才行。

将上述结果合并起来，可得

$$\mathrm{GP2LT}\left(\frac{2^n}{\sum_{i=0}^{d-2} \binom{n-1}{i}}\right) \leqslant A(n,d) \leqslant \frac{2^n}{\sum_{i=0}^{\lfloor d-1/2 \rfloor} \binom{n}{i}} \tag{6}$$

GP2LT 表示（严格）小于其参数值最大的 2 的幂。

表 15.7 列出了当 n 和 d 较小时由上述不等式所给出的取值范围。如果某单元格内仅有单个数字而非数字范围，那么表明由不等式（6）所给出的上下界相等。

表 15.7 $A(n,d)$ 的 G-V 界与汉明界

n	**d=4**	**d=6**	**d=8**	**d=10**	**d=12**	**d=14**	**d=16**	**n**
6	4−5	2	—	—	—	—	—	**5**
7	8−9	2	—	—	—	—	—	**6**
10	32−51	4−11	2−3	2	—	—	—	**9**
13	256−315	16−51	2−13	2−5	1	—	—	**12**
16	2048	64−270	8−56	2−16	2−6	2−3	2	**15**
19	8192−13797	256−1524	16−265	4−64	2−20	2−8	2−4	**18**
22	65536−95325	1024−9039	64−1342	8−277	4−75	2−25	2−10	**21**
25	2^{19}−671088	4096−55738	256−7216	32−1295	8−302	2−88	2−31	**24**
28	2^{22}−4793490	32768−354136	1024−40622	128−6436	16−1321	4−337	2−104	**27**
	d=3	**d=5**	**d=7**	**d=9**	**d=11**	**d=13**	**d=15**	

⊖ 此处“严格”(strictly) 一词的意思是，即便不等式右侧本身已经是 2 的幂了，也依然要向下调整。——译者注

若 d 为偶数，则可直接根据（6）式求出边界，也可以把（6）式左右两边所有的 d 与 n 都分别换为 $d-1$ 与 $n-1$ 之后再求解，因为根据（5）式可知，这样求解也可以。后一种办法求出的取值边界要么和前者相等，要么比其更“紧”（tighter）。因此，表 15.7 中的值[一]都是在 d 为奇数时求出的。如果要查 d 是偶数时的取值，那么可根据最顶行的 d 值与最左栏的 n 值来查。

（6）式给出的取值范围相当宽泛，在 d 较大时尤其如此。随着 n 值增大，上下界的比值发散为无穷大（diverge to infinity）。下界比上界还要宽泛。上千篇论文都在讲述如何改进取值范围，表 15.8 列出了编写本书时已知的研究成果［Agrell，Brou；当两者给出的取值范围不同时，表 15.8 列出了较为严格的那个］。

$(n,d)=(7,3)$、$(15,3)$、$(23,7)$ 的情况是“完美编码”（perfect code），也就是说，其取值范围达到了不等式（6）的上界。这一定义比 15.2 节所述更为通用。n 为奇数且 $n=d$ 时的编码也是完美编码，参见习题 8。

本章最后要说的是，整套编码的最小距离并非衡量二进制 FEC 码块优劣的唯一标准，也就是说，具备 p 位检测能力与 q 位错误更正能力的编码未必处处“强大”。例如，有人研究了旨在更正突发错误的编码。［Etzion］证明了（16,11）等编码既可更正单一位元错误，也可更正相邻两个位元中任何一个错误位元，并且是完美编码，这一研究主题不在本书讨论范围内。此类编码不具备一般意义上的双位元查错能力。而（16,11）扩展汉明码则既是 SEC-DED 码，也是完美编码。因此可以说，Etzion 的编码放弃了常说的双位元查错能力，而以之换取“连续双位元纠错能力”（double-bit error correction of consecutive bit）。这么做当然有意义，因为很多应用场景中，都会突然发生连续几个位元出错的情况（short burst）。

表 15.8 已知 $A(n,d)$ 的最精确取值范围

n	$d=4$	$d=6$	$d=8$	$d=10$	$d=12$	$d=14$	$d=16$	n
6	4	2	—	—	—	—	—	**5**
7	8	2	—	—	—	—	—	**6**
8	16	2	2	—	—	—	—	**7**
9	20	4	2	—	—	—	—	**8**
10	40	6	2	2	—	—	—	**9**
11	72	12	2	2	—	—	—	**10**
12	144	24	4	2	2	—	—	**11**
13	256	32	4	2	2	—	—	**12**
14	512	64	8	2	2	2	—	**13**
15	1024	128	16	4	2	2	—	**14**
16	2048	256	32	4	2	2	2	**15**
17	2720－3276	256－340	36	6	2	2	2	**16**
18	5312－6552	512－673	64－72	10	4	2	2	**17**
19	10496－13104	1024－1237	128－135	20	4	2	2	**18**

[一] 由最右栏的 n 和最底行的 d 所确定。——译者注

（续）

n	$d=4$	$d=6$	$d=8$	$d=10$	$d=12$	$d=14$	$d=16$	n
20	20480—26168	2048—2279	256	40	6	2	2	**19**
21	36864—43688	2560—4096	512	42—47	8	4	2	**20**
22	73728—87376	4096—6941	1024	64—84	12	4	2	**21**
23	147456—173015	8192—13674	2048	80—150	24	4	2	**22**
24	294912—344308	16384—24106	4096	128—268	48	6	4	**23**
25	2^{19}—599184	16384—47538	4096—5412	192—466	52—55	8	4	**24**
26	2^{20}—1198368	32768—84260	4101—9672	384—836	64—96	14	4	**25**
27	2^{21}—2396736	65536—157285	8192—17768	512—1585	128—169	28	6	**26**
28	2^{22}—472950	131072—291269	16384—32151	1024—3170	178—288	56	8	**27**
	$d=3$	$d=5$	$d=7$	$d=9$	$d=11$	$d=13$	$d=15$	

15.5 习题

1. 仿照表15.1，列出一套 $m=3$ 的汉明码。
2. 在特定的SEC码应用场景中，无需修复校验位中的错误。因此，k 个校验位只需校验信息位，而不需校验其本身。在有 m 个信息位的情况下，k 必须足够大，以便接收者能分辨出 $m+1$ 种情况：是 m 位中某个位传错了，还是传输无误。这样一来，所需校验位的数量就满足 $2^k \geqslant m+1$。与汉明规则相比，此不等式的限制更为宽泛，所以，对某些 m 值来说，似乎可以构建一套SEC码，令其校验位的个数比根据汉明规则构建出来的编码要少。还有一种办法，就是只用一个值来表示校验位里出错了，但是不指明具体是哪个校验位错了。于是，可得 $2^k \geqslant m+2$。

   上述推理错在何处？
3. （脑筋急转弯）给定 m 值，如何找出满足（1）式的最小 k 值？
4. 请证明：任何二进制码块的汉明距离函数都满足三角不等式，也就是说，将 $\boldsymbol{x}$，$\boldsymbol{y}$ 两个码串的汉明距离记为 $d(\boldsymbol{x},\boldsymbol{y})$，则有 $d(\boldsymbol{x},\boldsymbol{z}) \leqslant d(\boldsymbol{x},\boldsymbol{y}) + d(\boldsymbol{y},\boldsymbol{z})$。
5. 证明 $A(2n,2d) \geqslant A(n,d)$。
6. 描述"辛格顿界"（singleton bound）的不等式是：$A(n,d) \leqslant 2^{n-d+1}$，证明此式成立。
7. 不等式（6）的右侧若取等号，则说明此编码为完美编码，请证明：这一判定规则推广了汉明规则。
8. 如果 $n=d$，那么 $A(n,d)$ 的值是什么？然后证明：当 n 为奇数时，此类编码皆为完美编码。
9. 请证明：如果 n 是3的倍数且 $d=2n/3$，那么 $A(n,d)=4$。
10. 请证明：如果 $d>2n/3$，则 $A(n,d)=2$。
11. 有种二维奇偶校验法：把64个信息位 $\boldsymbol{u}_0 \ldots \boldsymbol{u}_{63}$ 排成 8×8 矩阵，然后依照下图为每行及每列添加一个奇偶校验位。

$$\begin{bmatrix} \boldsymbol{u}_0 & \boldsymbol{u}_1 & \cdots & \boldsymbol{u}_6 & \boldsymbol{u}_7 & \boldsymbol{r}_0 \\ \boldsymbol{u}_8 & \boldsymbol{u}_9 & \cdots & \boldsymbol{u}_{14} & \boldsymbol{u}_{15} & \boldsymbol{r}_1 \\ & & \cdots & & & \\ \boldsymbol{u}_{48} & \boldsymbol{u}_{49} & \cdots & \boldsymbol{u}_{54} & \boldsymbol{u}_{55} & \boldsymbol{r}_6 \\ \boldsymbol{u}_{56} & \boldsymbol{u}_{57} & \cdots & \boldsymbol{u}_{62} & \boldsymbol{u}_{63} & \boldsymbol{r}_7 \\ \boldsymbol{c}_0 & \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_6 & \boldsymbol{c}_7 & \boldsymbol{r}_8 \end{bmatrix}$$

$\boldsymbol{r}_i$是相关行的奇偶校验位，而 $\boldsymbol{c}_i$是相关列的奇偶校验位。右下角那个校验位可以校验各行的奇偶校验位，也可校验各列的奇偶校验位（但是不能两者都校验），本例将其设为最后一行的奇偶校验位（也就是校验 $\boldsymbol{c}_0$ 至 $\boldsymbol{c}_7$ 的整体奇偶性）。

请评价此方案。着重考虑如下问题：这套编码是不是 SEC-DED 码？将右下角 $\boldsymbol{r}_8$ 省去，能否显著改变其查错与纠错能力？右下角那一位可作为各行的总体校验位，亦可作为各列的总体校验位，这两种情况下的取值有何关系？

第 16 章

希尔伯特曲线

1890 年，朱塞佩·皮亚诺（Giuseppe Peano，1858—1932）发现了一种平面曲线（planar curve）[㊀]，该曲线的“空间填充”（space-filling）特性令人相当惊讶。

把单位正方形（unit square）的边三等分，于是就有了 9 个更小的正方形，“皮亚诺曲线”正是基于此而构建的。这种曲线按照一定顺序穿过 9 个小正方形。然后，每个小正方形也按类似方式划分为 9 个更小的正方形，同时修改曲线的样貌，使其能按一定顺序穿过这 9 个更小的正方形。此曲线可用三进制的分数来表示，实际上，皮亚诺起初正是这样描述它的。

1891 年，大卫·希尔伯特 [Hil] 发现了皮亚诺曲线的变体，他把单位正方形的每个边二等分，于是出现了 4 个小正方形，这种变体曲线是基于此法而构建的。每个小正方形也可按相似方式分为 4 个更小的正方形，依此类推。在每次向下细分的步骤中，希尔伯特都能画出一条穿越所有正方形的曲线。希尔伯特曲线是这种划分过程的“极限曲线”（limit curve），有时也叫“皮亚诺-希尔伯特曲线”（Peano-Hilbert curve）。此曲线可表示为二进制分数。

图 16.1 按照 1891 年的论文演示了“希尔伯特空间填充曲线”（Hilbert space-filling curve）的前 3 步。

本书略作修改。假如有一系列曲线，其极限是希尔伯特空间填充曲线，那么，我们把其中任何一条曲线都称为“希尔伯特曲线”。序列中的第 n 条曲线就叫“n 阶希尔伯特曲线”。图 16.1 分别画出了 1 阶、2 阶、3 阶希尔伯特曲线。将这些曲线向左下方移动，使其每个拐角处恰好和划分正方形的那些横竖线所构成的交点重合。最后，把 n 阶曲线放大为原来的 2^n 倍，这样的话，每个拐角的坐标就都是整数了。于是，n 阶希尔伯特曲线的拐角所在的横、纵坐标均在 0 至 2^n-1 范围内。我们把曲线 (x,y) 的坐标从（0，0）变为（2^n-1，0）的方向当做正向。图 16.2 列出了 1 阶至 6 阶的希尔伯特曲线。

㊀ 曲线是由一维空间至多维空间的连续映射。

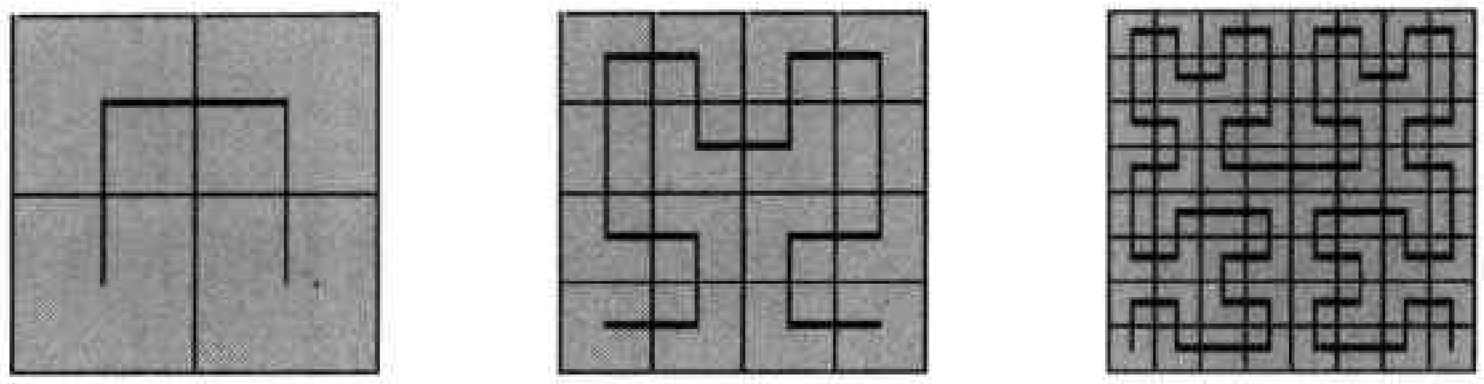


图 16.1 希尔伯特曲线序列中的前3条曲线

16.1 生成希尔伯特曲线的递归算法

观察图 16.2 便可明白如何生成希尔伯特曲线了。1 阶曲线从左下角开始，向上，向右，再向下。2 阶曲线的总体形式也如此。首先，画一条总体方向[一]往上的左开口 U 形曲线。然后，向上画一条单位长度的线段。第三步，画一条总体方向往右的倒 U 形曲线，向右画一条单位线段，再画一个总体方向往右的倒 U 形曲线。最后，向下画一条单位线段，再画一条总体方向往下的右开口 U 形曲线。

按照上面步骤，即可把 1 阶倒 U 形曲线化为 2 阶 Y 形曲线。

可以认为，任何阶数的希尔伯特曲线都由一系列不同方向的 U 形曲线所组成，除了最后一个 U 形曲线外，其余每个曲线后面都跟着一个单位线段。要将某阶数的希尔伯特曲线化为其下一阶，可以把当前这一阶中的每个 U 形曲线都按上述方法化为下一阶中的 Y 形曲线，并保持两者大方向不变，同时，也要把这一阶中的单位线段化为下一阶中方向相同的单位线段。

按如下步骤执行，即可将 1 阶希尔伯特曲线（也就是“总体方向”往右且顺时针旋转[二]的 U 形曲线[三]）变换为 2 阶：

1. 画一条总体方向往上且逆时针旋转的 U 形曲线。
2. 向上画单位线段。
3. 画一条总体方向往右且顺时针旋转的 U 形曲线。
4. 向右画单位线段。
5. 画一条总体方向往右且顺时针旋转的 U 形曲线。
6. 向下画单位线段。
7. 画一条总体方向往下且逆时针旋转的 U 形曲线。

[一] 原文为“总体方向往上”（go up，in net effect）。1 阶曲线（即各种开口方向的简单 U 形曲线）要画 3 笔，“总体方向”就是第 2 笔的走向。对于高阶曲线来说，如果从左下角出发，经由上方绕至右下角，则其“总体方向”为“右”，余可类推。——译者注

[二] 对于 1 阶曲线来说，“旋转方向”就是从第 1 笔到第 2 笔、从第 2 笔到第 3 笔的转向。例如一条开口往下且第 2 笔向右的 U 形曲线，其“旋转方向”就是“顺时针旋转 90 度”。对于高阶曲线来说，如果从左下角出发，经由上方绕至右下角，则其“旋转方向”为“顺时针旋转 90 度”，余可类推。——译者注

[三] 也可理解为第 2 笔向右的“倒 U 形曲线”。——译者注

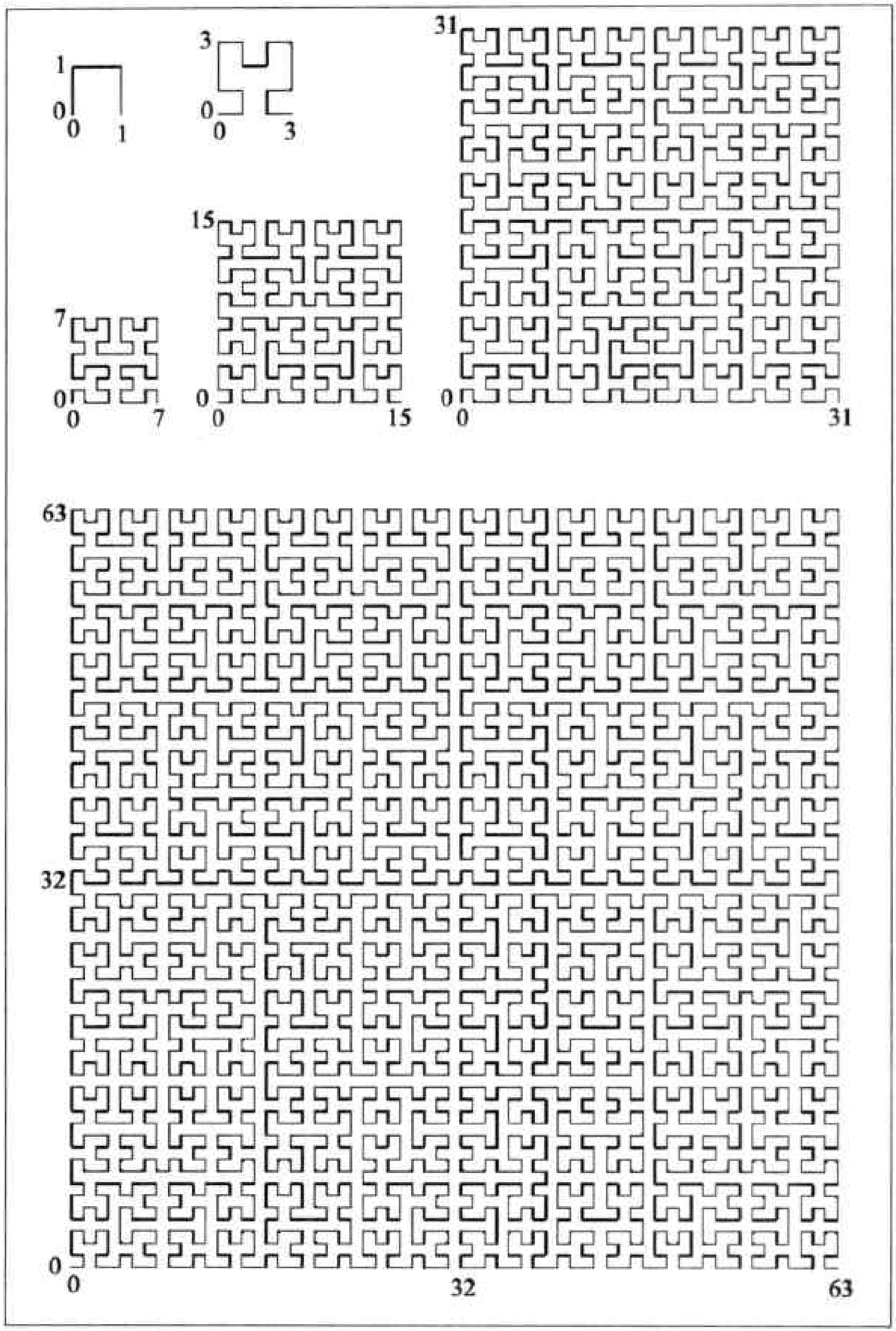


图 16.2 1 至 6 阶希尔伯特曲线

经检验可知，所有与 1 阶希尔伯特曲线“方向”㊀相同的高阶曲线都可由这一方式变换出来。想要把其他走向的 U 形曲线变换为相应高阶曲线，也有一套类似规则。图 16.3 [Voor] 中的递归程序码体现了这些规则。此程序用两个整数来决定如何绘制希尔伯特曲

㊀ 这里“方向”一词是“总体方向”（net linear direction）与“旋转方向”（rotational direction）的合称。——译者注

线内的各条U形曲线，其中一个（dir）表示希尔伯特曲线的“总体方向”，另一个（rot）表示“旋转方向”，其参数的意义如下：

dir = 0：右	rot = +1：顺时针旋转 90 度
dir = 1：上	rot = −1：逆时针旋转 90 度
dir = 2：左	
dir = 3：下	

实际上，dir 也可以取其他值，只要保证那些值除以 4 的余数和上面这套取值相符即可。

```
void step(int);

void hilbert(int dir, int rot, int order) {

   if (order == 0) return;

   dir = dir + rot;
   hilbert(dir, -rot, order - 1);
   step(dir);
   dir = dir - rot;
   hilbert(dir, rot, order - 1);
   step(dir);
   hilbert(dir, rot, order - 1);
   dir = dir - rot;
   step(dir);
   hilbert(dir, -rot, order - 1);
}
```

图 16.3　希尔伯特曲线生成器

图 16.4 列出了演示用的驱动程序（driver program）[⊖] 以及 hilbert 函数中用到的 step 函数。把待构建的希尔伯特曲线所具备的阶数告诉该程序，就能输出一个线段列表，其中给出了每次移动的方向、由曲线起点到当前线段终点所途经的长度，以及当前线段终点的坐标。例如，2 阶曲线的输出结果就是：

```
 0   0000   00 00
 0   0001   01 00
 1   0010   01 01
 2   0011   00 01
 1   0100   00 10
 1   0101   00 11
 0   0110   01 11
-1   0111   01 10
 0   1000   10 10
 1   1001   10 11
 0   1010   11 11
-1   1011   11 10
-1   1100   11 01
-2   1101   10 01
-1   1110   10 00
 0   1111   11 00
```

⊖ 是指为了演示某个算法或功能而编写的测试程序，与常说的硬件驱动程序不是同一个概念。——译者注

```c
#include <stdio.h>
#include <stdlib.h>

int x = -1, y = 0;                 // Global variables.
int s = 0;                         // Dist. along curve.
int blen;                          // Length to print.

void hilbert(int dir, int rot, int order);

void binary(unsigned k, int len, char *s) {
/* Converts the unsigned integer k to binary character
form. Result is string s of length len. */
   int i;

   s[len] = 0;
   for (i = len - 1; i >= 0; i--) {
      if (k & 1) s[i] = '1';
      else       s[i] = '0';
      k = k >> 1;
   }
}
void step(int dir) {
   char ii[33], xx[17], yy[17];

   switch(dir & 3) {
      case 0: x = x + 1; break;
      case 1: y = y + 1; break;
      case 2: x = x - 1; break;
      case 3: y = y - 1; break;
   }
   binary(s, 2*blen, ii);
   binary(x, blen, xx);
   binary(y, blen, yy);
   printf("%5d   %s   %s %s\n", dir, ii, xx, yy);
   s = s + 1;                      // Increment distance.
}
int main(int argc, char *argv[]) {
   int order;

   order = atoi(argv[1]);
   blen = order;
   step(0);                        // Print init. point.
   hilbert(0, 1, order);
   return 0;
}
```

图 16.4 希尔伯特曲线生成器的演示程序

16.2 根据希尔伯特曲线上从起点到某点的途经距离求其坐标

n 阶希尔伯特曲线上有一个点，从起点出发，沿着曲线到达该点所经距离为 s，如果想求此点的（x,y）坐标，可以把 s 写为二进制整数，然后根据权重最高的两个位元就能判断出这个点位于哪个“象限”（quadrant，四分之一区域）[⊖]。其原因在于，对于任意阶数的希尔伯特曲线，其整体形式都与 1 阶曲线相同。最高两位若是 00，则这个点必在左

⊖ 这里“quadrant”一词与常用含义稍有不同，此处是指包围希尔伯特曲线的大正方形中左下、左上、右上、右下这 4 个区域。并不完全等同于平面直角坐标系中的象限，译文只是借用该词，故加引号。——译者注

下方的某个位置上；若是 01，则必在左上方；若是 10，则在右上方；若是 11，则在右下方。因此，s 的最高两位决定了 n 位整数 x 与 y 的最高有效位。

s 的最高两位	(x, y) 的最高有效位
00	(0，0)
01	(0，1)
10	(1，1)
11	(1，0)

理论上有 8 种 U 形曲线的画法⊖，而实际上希尔伯特曲线只会用到其中 4 种。表 16.1 以图示和映射的形式列出了如何根据两位二进制数 s 来推出 1 位二进制数 x 与 y 的取值。

表 16.1　可能会出现的 4 类映射关系

A	B	C	D
00→(0，0)	00→(0，0)	00→(1，1)	00→(1，1)
01→(0，1)	01→(1，0)	01→(1，0)	01→(0，1)
10→(1，1)	10→(1，1)	10→(0，0)	10→(0，0)
11→(1，0)	11→(0，1)	11→(1，1)	11→(10)

由图 16.2 可知，所有符合 A 类映射（）的 U 形曲线在向下一阶变换的过程中，都会“分裂”成 4 条 U 形曲线，从原 U 形曲线的起点算起，沿着曲线往下走，这 4 条曲线新分裂出来的曲线类型依次为 B、A、A、D，其“分裂点”距离原曲线起点的距离分别为 0、1、2、3。同理，当前这一阶里的 B 类映射（）在下一阶中，也成了 4 条新的 U 形曲线，其类型分别为 A、B、B、C，从原曲线的角度来说，这 4 条新曲线与原曲线起点的距离分别为 0、1、2、3。

可将这些论述归纳为 16.2 中的状态转换表（state transition table），其中每种“状态”都对应于表 16.1 中相关的映射类型。

要使用此表，需从状态 A 开始。设希尔伯特曲线的阶数为 n，给二进制整数 s 左方添 0，使其长度变为 $2n$。从左至右，每次取出 s 中的两个位元。表 16.2 第一行的意思是，如果当前状态是 A，且从 s 里取出的两位是 00，那么就输出（0，0），并切换至状态 B。然后再从 s 里取出下面两个位元。同理，第二行的意思是，若当前状态为 A，且从 s 里取出的两位是 01，那么输出（0，1），并继续保持状态 A。

⊖ 在上一节所讲的 hilbert 函数中，第 1 个参数 dir 有 4 种取值，第 2 个参数 rot 有两种取值，所以二者共有 4×2=8 种取值组合，这也就表示此处所说的 8 种画法。——译者注

表 16.2 计算 (x,y) 所用的状态转换表

若当前处于状态	且从 s 里（由左至右）取出的两位是	则在 (x,y) 两数右方分别追加	并进入状态
A	00	(0, 0)	B
A	01	(0, 1)	A
A	10	(1, 1)	A
A	11	(1, 0)	D
B	00	(0, 0)	A
B	01	(1, 0)	B
B	10	(1, 1)	B
B	11	(0, 1)	C
C	00	(1, 1)	D
C	01	(1, 0)	C
C	10	(0, 0)	C
C	11	(0, 1)	B
D	00	(1, 1)	C
D	01	(0, 1)	D
D	10	(0, 0)	D
D	11	(1, 0)	A

把每次输出的位元从左至右收集起来。当 s 中的位元全部测试完毕后，所输出的两个 n 位二进制数就是 x、y 的值。

举例来说，假设 $n=3$ 且

$$s=11\ 0100$$

运算过程由状态 A 开始，s 头两位是 11，于是，输出（1，0），并切换至 D 状态（根据第 3 行第 4 列查得）。然后，在 D 状态下取出 s 后两位 01，根据表格第 14 行，输出（0，1）并继续留在 D 状态。最后，根据表格第 13 行，输出（1，1）并切换至 C 状态，当然，这个状态其实用不到。

因此，输出结果是（101，011），化为十进制，就是 $x=5$ 且 $y=3$。

图 16.5 用 C 语言实现了上述步骤。程序用 0 至 3 来表示当前状态，这 4 个整数分别对应于 A 至 D 这 4 种状态。程序把当前状态同 s 的下面两位拼接起来，并赋给变量 `row`，于是 `row` 值就成了 0 至 15 之间的整数，据此即可查到表 16.2 中的对应行。程序把表 16.2 右侧两行从下到上编码成 3 个位串，并将其写为十六进制数，然后用变量 `row` 来查[⊖]，也就是说，此处使用内存查表法。如果把这 3 个十六进制数展开为二进制位串，并从左往右看，那么就能发现其分别对应表 16.2 中从下至上每行里所列的 x、y 及状态值。

⊖ 例如，将右侧第二列的 x 坐标从下至上写出来，就是 1001 0011 0110 1100，变为十六进制则是 0x936C。在位串中，左侧首个位元 1 对应 `row` 值为 15 的情况，左侧第二个位元 0 对应 `row` 值为 14 的情况……最右侧位元 0 对应 `row` 值为 0 的情况，所以程序使用（0x936C≫row）& 1 这种内存查表法来查得其值。0x39C6、0x3E6B 94C1 这两个十六进制常数同理可得。——译者注

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                      unsigned *yp) {

   int i;
   unsigned state, x, y, row;

   state = 0;                             // Initialize.
   x = y = 0;

   for (i = 2*n - 2; i >= 0; i -= 2) {   // Do n times.
      row = 4*state | (s >> i) & 3;       // Row in table.
      x = (x << 1) | (0x936C >> row) & 1;
      y = (y << 1) | (0x39C6 >> row) & 1;
      state = (0x3E6B94C1 >> 2*row) & 3; // New state.
   }
   *xp = x;                               // Pass back
   *yp = y;                               // results.
}
```

图 16.5 根据 s 计算 (x,y)

[L&S] 给出了一个与上例区别很大的算法。与图 16.5 所示算法不同，此算法从右至左扫描 s。此算法的思路是：先从 s 中取出最低两位，根据 1 阶希尔伯特曲线将其映射为 (x, y)，然后再向左取出两位，继续测试。如果第二次取出的两位是 00，那么要把刚计算出的 x 与 y 交换，因为原来的 1 阶曲线和现在 2 阶曲线左下角的那块图形关于 $x=y$ 这条直线对称（参见图 16.1 中的 1 阶和 2 阶曲线）。如果取出的是 01 或 10，那么 x、y 均不变。如果是 11，那么要把 x、y 交换并各自求补。若由左至右处理 s 中的位元，则此套规则亦适用。表 16.3 与图 16.6 均体现了这些规则。令人稍觉惊奇的是：还可以先把位元添在 x、y 前面，然后连着新加入的位元一起交换和求补，这样算出来的结果也正确。

表 16.3 按照“Lam and Shapiro 算法”由 s 求出 (x,y)

如果 s（从右至左）下面两位是	那么	并在 (x,y) 前面添加
00	交换 x、y	(0, 0)
01	不变	(0, 1)
10	不变	(1, 1)
11	交换 x、y 并各自求补	(1, 0)

图 16.6 并未初始化 x 与 y，某些编译器可能会给出错误信息，不过无论 x、y 初始值是什么，代码总能算出正确结果。

按 2.20 节提到的“三次异或”技巧改写代码，就能去掉图 16.6 循环中的分支语句。可用下列语句替换掉 if 语句块，其中 swap 和 cmpl 均是无符号整数：

```
swap = (sa ^ sb) - 1;  // -1 if should swap, else 0.
cmpl = -(sa & sb);     // -1 if should compl't, else 0.
x = x ^ y;
y = y ^ (x & swap) ^ cmpl;
x = x ^ y;
```

改写后需要 9 条指令，而改写前的 if 语句块需要执行两条或 6 条指令，所以说，改写后的代码只有在分支语句开销很大时才能发挥优势。

根据 [L&S] 的“交换并求补”（swap and complement）这一思路，可研发一种生

成希尔伯特曲线的逻辑电路。下述电路的原理是：绘制 n 阶曲线的过程实际上就是根据表 16.1 的 A 类映射，把 s 里的位元对化为（x，y）值。当曲线途经各个区域时，有可能要把刚才求出的映射值交换或求补，也有可能既要交换又要求补。图 16.7 中的电路根据上一轮所输出的信号得知当前这轮是否需要执行交换与求补操作，并依此把 s 中的两个位元映射为相应的（x_i，y_i）值，然后生成下一轮所用的交换和求补信号。

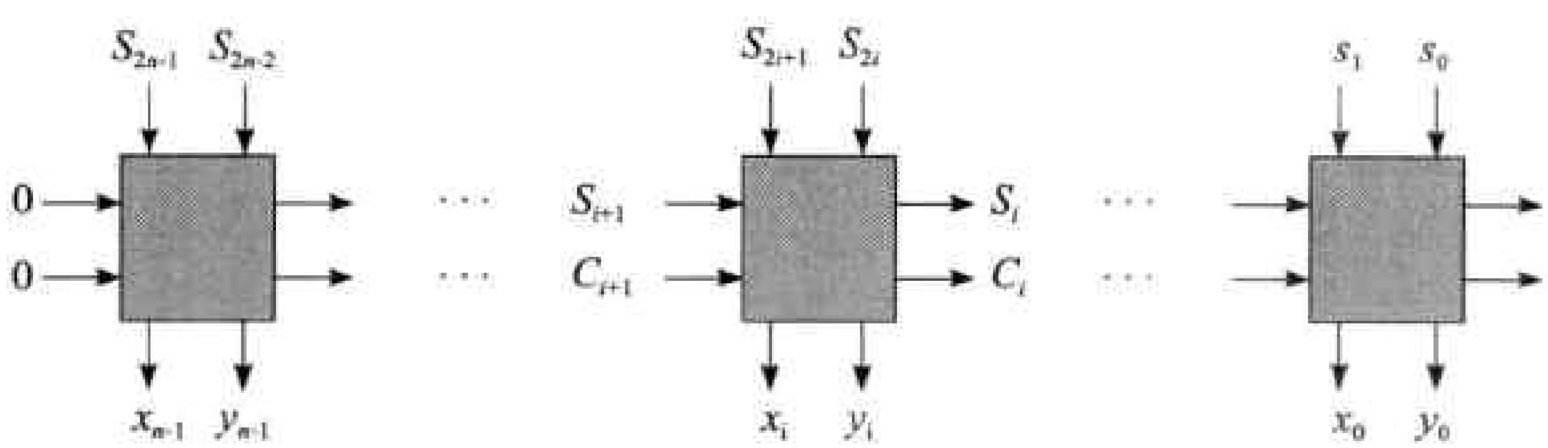
```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                       unsigned *yp) {

   int i, sa, sb;
   unsigned x, y, temp;

   for (i = 0; i < 2*n; i += 2) {
      sa = (s >> (i+1)) & 1;      // Get bit i+1 of s.
      sb = (s >> i) & 1;          // Get bit i of s.

      if ((sa ^ sb) == 0) {       // If sa,sb = 00 or 11,
         temp = x;                // swap x and y,
         x = y^(-sa);             // and if sa = 1,
         y = temp^(-sa);          // complement them.
      }
      x = (x >> 1) | (sa << 31);  // Prepend sa to x and
      y = (y >> 1) | ((sa ^ sb) << 31); // (sa^sb) to y.
   }
   *xp = x >> (32 - n);           // Right-adjust x and y
   *yp = y >> (32 - n);           // and return them to
}                                 // the caller.
```

图 16.6　根据"Lam and Shapiro 算法"由 s 求出（x,y）



$$x_i = [s_{2i+1} \oplus (s_{2i} S_{i+1})] \oplus C_{i+1}$$
$$y_i = [s_{2i} \oplus S_{2i+1} \oplus (s_{2i} S_{i+1})] \oplus C_{i+1}$$
$$S_i = S_{i+1} \oplus (s_{2i} = S_{2i+1})$$
$$C_i = C_{i+1} \oplus (s_{2i} S_{2i+1})$$

图 16.7　使用逻辑电路，由 s 求出（x,y）

假设某寄存器中包含路径长度 s 以及递增 s 所用的电路。那么，想找出希尔伯特曲线的下一个点，首先要递增 s，然后按照表 16.4 将其变换为（x,y）。此处有个小麻烦：变换过程是从左向右执行的，而递增 s 的操作却是个从右至左的过程。于是，生成 n 阶希尔伯特曲线上的一个新点所需的时间和 $2n+n$ 成正比，其中 $2n$ 对应于递增 s 所需的时间，而 n 对应于将 s 变换成（x,y）所需的时间。

图 16.7 将此计算过程表示为逻辑电路。图中的 S 代表交换信号，C 代表求补信号。

图16.7中的逻辑电路演示了另外一种根据 s 来计算 (x,y) 的办法。观察交换与求补信号是如何在这 n 个步骤中由左至右传递的。由此可知，应该能使用并行前缀操作，迅速将交换与求补信息传播至各个阶段中（这样只需 $\log_2 n$ 步，而非 $n-1$ 步），然后按照图16.7中的等式，执行某些字组级别的逻辑操作，以此计算 x 与 y。x 与 y 值交织排列于字组中的偶数与奇数位置上，所以得用“理牌操作”（unshuffle operation，参见7.2节）将其分开。这似乎有点复杂，而且可能只在 n 相当大时才能发挥其优势，不过，我们还是来看看如何实现此算法。

表16.4 由 s 计算 (x,y) 的逻辑

如果 s（从左至右）下面两位是	则在 (x,y) 后面追加	并设置
00	(0，0)①	swap $=\overline{\text{swap}}$
01	(0，1)①	不变
10	(1，1)①	不变
11	(1，0)①	$\overline{\text{swap}}=$，cmpl $=\overline{\text{cmpl}}$

①有可能要执行交换及/或求补操作

图16.8列出了此算法的一种实现代码［GLS1］。由于这段程序要在全字上操作，所以先给 s 左边填充若干对“01”位元。这种位元组合不影响交换与求补标志的值。然后计算出 cs 的值（cs 表示 complement-swap，求补-交换）。字组中的位元以 cscs...cs 的形式出现，其中每个 c 位表示相应的位元对是否应求补，若是1则需求补，而每个 s 位则表示相应的位元对是否应该交换，求补与交换的规则参见表16.3。换句话说，这两行语句按照下列表格把 s 中的每对位元映射为 cs 中的两位。

s_{2i+1}	s_{2i}	cs
0	0	01
0	1	00
1	0	00
1	1	11

上表中的值描述了我们想运用并行前缀操作而执行的运算。应该选用 PP-XOR（并行前缀异或）操作，因为从左往右看，连续两个值为1的求补信号或连续两个值为1的交换信号，其意义都与异或操作的逻辑属性相同：连续出现的两个1会互相抵消。

两种信号（求补与交换）都在同一个 PP-XOR 操作中传播，二者交错排布在 cs 变量的各个位置上。

接下来4条赋值语句会将 s 中的每对位元转译为（x，y）值，其中 x 放在奇数位置上，y 放在偶数位置上（左起首位算作1号位）。虽说其中的逻辑看上去似乎有些晦涩，然而不难验证，s 中每对位元都是按图16.7前两个布尔等式来变换的。（提示：由 t 与 sr 的奇数位置均为0这一事实出发，分别思考偶数位置与奇数位置是如何变换的。）

接下来的程序无需说明。此程序共执行66条基本 RISC 指令（此值恒定，这些指令中也没有分支指令），而图16.6中的代码平均要执行 $19n+10$ 条指令（此指令数乃根据代

码编译后的汇编码推算而得，统计时也将“函数序言”(prolog) 和“函数尾声”(epilog) 考虑在内，不过本例中实际上没有这两部分)。于是可知，在 $n \geqslant 3$ 时，并行前缀算法更快。

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                      unsigned *yp) {
   unsigned comp, swap, cs, t, sr;

   s = s | (0x55555555 << 2*n); // Pad s on left with 01
   sr = (s >> 1) & 0x55555555;  // (no change) groups.
   cs = ((s & 0x55555555) + sr) // Compute complement &
        ^ 0x55555555;           // swap info in two-bit
                                // groups.
   // Parallel prefix xor op to propagate both complement
   // and swap info together from left to right (there is
   // no step "cs ^= cs >> 1", so in effect it computes
   // two independent parallel prefix operations on two
   // interleaved sets of sixteen bits).

   cs = cs ^ (cs >> 2);
   cs = cs ^ (cs >> 4);
   cs = cs ^ (cs >> 8);
   cs = cs ^ (cs >> 16);
   swap = cs & 0x55555555;      // Separate the swap and
   comp = (cs >> 1) & 0x55555555;  // complement bits.

   t = (s & swap) ^ comp;       // Calculate x and y in
   s = s ^ sr ^ t ^ (t << 1);   // the odd & even bit
                                // positions, resp.
   s = s & ((1 << 2*n) - 1);    // Clear out any junk
                                // on the left (unpad).

   // Now "unshuffle" to separate the x and y bits.

   t = (s ^ (s >> 1)) & 0x22222222; s = s ^ t ^ (t << 1);
   t = (s ^ (s >> 2)) & 0x0C0C0C0C; s = s ^ t ^ (t << 2);
   t = (s ^ (s >> 4)) & 0x00F000F0; s = s ^ t ^ (t << 4);
   t = (s ^ (s >> 8)) & 0x0000FF00; s = s ^ t ^ (t << 8);

   *xp = s >> 16;               // Assign the two halves
   *yp = s & 0xFFFF;            // of t to x and y.
}
```

图 16.8 使用并行前缀算法，由 s 求出 (x,y)

16.3 根据希尔伯特曲线上的坐标求从起点到某点的途经距离

给定希尔伯特曲线上某点坐标，可据此求出由起点至该点的途经距离，计算过程会用到一张与表 16.2 相似的状态转换表。表 16.5 正是这样一张表。

表 16.5 由 (x,y) 计算 s 时所用的状态转换表

若当前处于状态	且从 (x,y) 里（由左至右）取出的两位是	则在 s 右方追加	并进入状态
A	(0, 0)	00	B
A	(0, 1)	01	A
A	(1, 0)	11	D
A	(1, 1)	10	A
B	(0, 0)	00	A

（续）

若当前处于状态	且从 (x,y) 里（由左至右）取出的两位是	则在 s 右方追加	并进入状态
B	(0, 1)	11	C
B	(1, 0)	01	B
B	(1, 1)	10	B
C	(0, 0)	10	C
C	(0, 1)	11	B
C	(1, 0)	01	C
C	(1, 1)	00	D
D	(0, 0)	10	D
D	(0, 1)	01	D
D	(1, 0)	11	A
D	(1, 1)	00	C

转换过程与上一节所讲的相似。首先，应为 x、y 左侧添加前导 0，使二者均包含 n 个位元，其中 n 是希尔伯特曲线的阶数。第二步，由左至右扫描 x、y 里的位元，并由左至右构建出 s。

图 16.9 列出了一段可实现上述步骤的 C 语言程序。

```
unsigned hil_s_from_xy(unsigned x, unsigned y, int n) {

   int i;
   unsigned state, s, row;

   state = 0;                            // Initialize.
   s = 0;

   for (i = n - 1; i >= 0; i--) {
      row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
      s = (s << 2) | (0x361E9CB4 >> 2*row) & 3;
      state = (0x8FE65831 >> 2*row) & 3;
   }
   return s;
}
```

图 16.9 由 (x,y) 求 s 的算法

上一节说过，[L&S] 给出了一个从 s 求 (x,y) 的算法，与之类似，他们也发明了一个从 (x,y) 求 s 的算法。如表 16.6 与图 16.10 所示，这是个由左至右执行的算法。

表 16.6 按照“Lam and Shapiro 算法”由 (x,y) 求出 s

如果 (x,y)（从右至左）下面两位是	那么	并在 s 前面添加
(0, 0)	交换 x、y	00
(0, 1)	不变	01
(1, 0)	交换 x、y 并各自求补	11
(1, 1)	不变	10

```
unsigned hil_s_from_xy(unsigned x, unsigned y, int n) {

   int i, xi, yi;
   unsigned s, temp;

   s = 0;                          // Initialize.
   for (i = n - 1; i >= 0; i--) {
      xi = (x >> i) & 1;           // Get bit i of x.
      yi = (y >> i) & 1;           // Get bit i of y.

      if (yi == 0) {
         temp = x;                 // Swap x and y and,
         x = y^(-xi);              // if xi = 1,
         y = temp^(-xi);           // complement them.
      }
      s = 4*s + 2*xi + (xi^yi);    // Append two bits to s.
   }
   return s;
}
```

图 16.10 根据“Lam and Shapiro 算法”由（x，y）求出 s

16.4 递增希尔伯特曲线上点的坐标

给定 n 阶希尔伯特曲线上某点坐标（x,y），如何求出下一个点的坐标呢？一种办法是，根据前面所述各算法，把（x,y）转为 s，给 s 加 1，再把新 s 转回（x,y）。

还有个稍微好点（但并未大幅改观）的做法，它基于如下事实：沿着希尔伯特曲线向前走时，坐标会出现两种变化方式，要么下一步的 x 不变，而 y 值比上一步多 1 或少 1，要么下一步的 y 不变，而 x 值比上一步多 1 或少 1，这两种情况必会发生其一，而又不能同时发生。接下来要讲的算法是，由左至右扫描坐标，以决定当前两个位元对应哪一种 U 形曲线。然后，根据 U 形曲线的种类及这两个位元值，决定应该增减 x 还是增减 y。

基本思路如此，不过到了某段 U 形曲线最后一个点时，会有点小麻烦（每执行 4 步就要遇到一次这种情况）。此时要由 x 和 y 前面的位元查出哪些位在下一阶的曲线里对应何种 U 形曲线，然后结合那些位及下一阶中 U 形曲线的种类来决定怎样修改坐标。若该点也在下一阶中 U 形曲线的末端，则前面那两位与 U 形曲线的类型就指明了下一步的方向，其余情况亦可按此类推。

表 16.7 描述了此算法。其中 A、B、C、D 对应于 16.2 节表 16.1 所示 4 种 U 形曲线。使用此表前，首先为 x 与 y 添加前导 0，使其均变为 n 位，其中 n 是希尔伯特曲线的阶数。由 A 状态开始，由左至右扫描 x 与 y。表 16.7 第 1 行的意思是：若当前状态是 A，且扫描到的两位是（0，0），那么就把“递增 y”这一操作记录到某变量中，然后切换至 B 状态。其他各行也可类似解读，前面的负号表示递减相应坐标。第 3 栏若为破折号，则不要改动记录坐标变化方式的那个变量。

扫描完 x 与 y 的最末一位（也就是最右侧位元）之后，按照变量的最终值来增减相应坐标。

图 16.11 用 C 语言实现了上述步骤。如果按照此方式初始化变量 dx，那么只要持续

调用该函数，算法就能反复生成同一条希尔伯特曲线[⊖]。（不过，希尔伯特曲线终点与起点的间距未必是 1。

表 16.7 计算希尔伯特曲线下一个点的坐标所用的表

若当前处于状态	且从 (x,y) 里（由左至右）取出的两位是	那么就准备递增/递减	并进入状态
A	(0, 0)	y^+	B
A	(0, 1)	x^+	A
A	(1, 0)	—	D
A	(1, 1)	y^-	A
B	(0, 0)	x^+	A
B	(0, 1)	—	C
B	(1, 0)	y^+	B
B	(1, 1)	x^-	B
C	(0, 0)	y^+	C
C	(0, 1)	—	B
C	(1, 0)	x^-	C
C	(1, 1)	y^-	D
D	(0, 0)	x^+	D
D	(0, 1)	y^-	D
D	(1, 0)	—	A
D	(1, 1)	x^-	C

```
void hil_inc_xy(unsigned *xp, unsigned *yp, int n) {

   int i;
   unsigned x, y, state, dx, dy, row, dochange;

   x = *xp;
   y = *yp;
   state = 0;                    // Initialize.
   dx = -((1 << n) - 1);         // Init. -(2**n - 1).
   dy = 0;

   for (i = n-1; i >= 0; i--) {          // Do n times.
      row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
      dochange = (0xBDDB >> row) & 1;
      if (dochange) {
         dx = ((0x16451659 >> 2*row) & 3) - 1;
         dy = ((0x51166516 >> 2*row) & 3) - 1;
      }
      state = (0x8FE65831 >> 2*row) & 3;
   }
   *xp = *xp + dx;
   *yp = *yp + dy;
}
```

图 16.11 用代码算出希尔伯特曲线的下一个点的坐标

很容易就能把表 16.7 实现为图 16.12 这样的逻辑电路。图中变量含义如下。

⊖ 可以理解为：如果把希尔伯特曲线的终点输入该函数，那么算法就将绕回到起点。——译者注

x_i： 输入值 x 的第 i 位

y_i： 输入值 y 的第 i 位

X，Y： 根据 S_{i+1} 与 C_{i+1}，交换并求补之后的新 x_i 与新 y_i

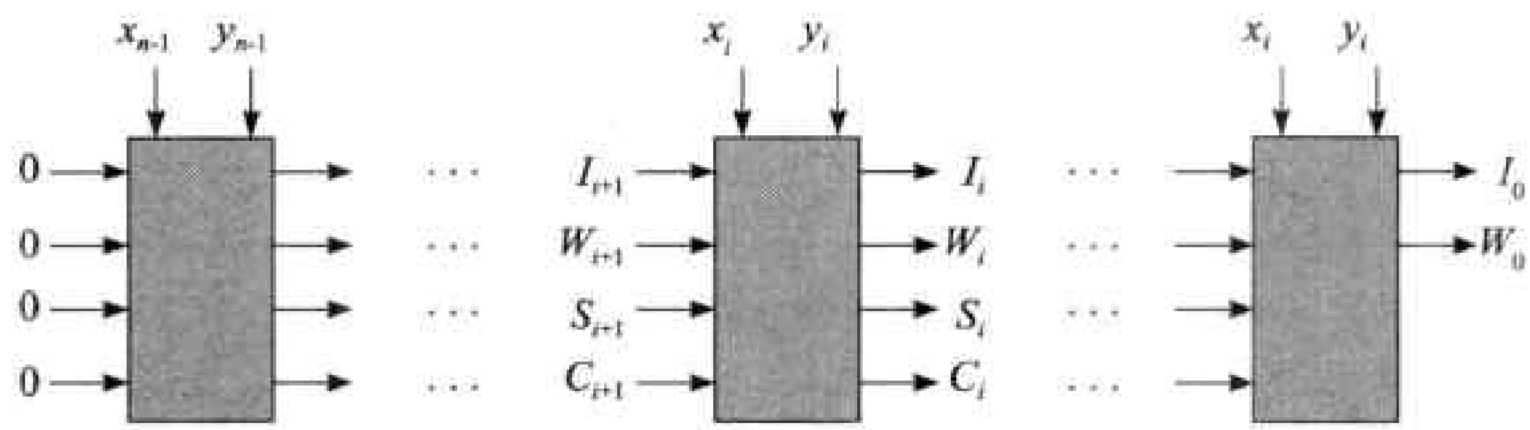
I： 若为 1 则表示递增 1；为 0 表示递减 1

W： 若为 1 则表示递增或递减 x；为 0 则表示递增或递减 y

S： 若为 1 则交换 x_i 与 y_i

C： 若为 1 则对 x_i 与 y_i 求补

S 与 C 合起来描述表 16.7 中的“状态”，$(C,S)=(0,0)$，$(0,1)$，$(1,0)$，$(1,1)$ 分别代表 A，B，C，D 这 4 种状态。输出信号 I_0 表示应该加 1 还是减 1，而 W_0 则表示要加减其值的坐标是 x 还是 y。（除了图中所示逻辑电路外，还需要一个递增器或递减器，并用数据选择器（MUX）将 x 或 y 输入给递增器或递减器，然后用电路把修改后的 x 与 y 存回其所在的寄存器中。另外，也可以用两个递增器或递减器电路来实现上述操作。）



$$X=(S_{i+1}y_i+\overline{s_{i+1}}x_i)\oplus C_{i+1}$$
$$Y=(S_{i+1}x_i+\overline{S_{i+1}}y_i)\oplus C_{i+1}$$
$$I_i=\overline{C_{i+1}}\,\overline{X}+C_{i+1}XY+I_{i+1}X\overline{Y}$$
$$W_i=\overline{S_{i+1}}\,\overline{X}Y+S_{i+1}(X\equiv Y)+W_{i+1}X\overline{Y}$$
$$S_i=S_{i+1}\equiv Y$$
$$C_i=C_{i+1}\oplus(X\overline{Y})$$

图 16.12 用逻辑电路算出希尔伯特曲线下一个点的 (x, y) 坐标

16.5 非递归的曲线生成算法

表 16.2 与表 16.7 描述了两种非递归算法，可生成任意阶数的希尔伯特曲线。这两种算法用硬件实现起来都不算太难。如果基于表 16.2，那么硬件算法就要用寄存器存放 s，并且在每一步中递增其值，然后转换为 (x,y) 坐标。如果基于表 16.7，那么不需要用寄存器来存放 s，但是算法要复杂一些。

16.6 其他空间填充曲线

如前所述，皮亚诺在 1890 年首次发现了空间填充曲线，其后发现的变种通常都称为

“皮亚诺曲线”（Peano curve）。以利亚金·黑斯廷斯·摩尔（Eliakim Hastings Moore，1862—1932）于1900年发现了一种有趣的希尔伯特曲线变体。这是一种“循环”曲线，也就是说，其终点与起点的间距为一个单位长度。图16.13演示了3阶皮亚诺曲线与4阶摩尔曲线。摩尔曲线有个特殊的地方，那就是1阶曲线中会出现“向上一向右一向下”的倒U形曲线（┌┐），而高阶曲线里却不再出现这一形状了。除了这个小例外，摩尔曲线的相关算法都和希尔伯特曲线非常类似。

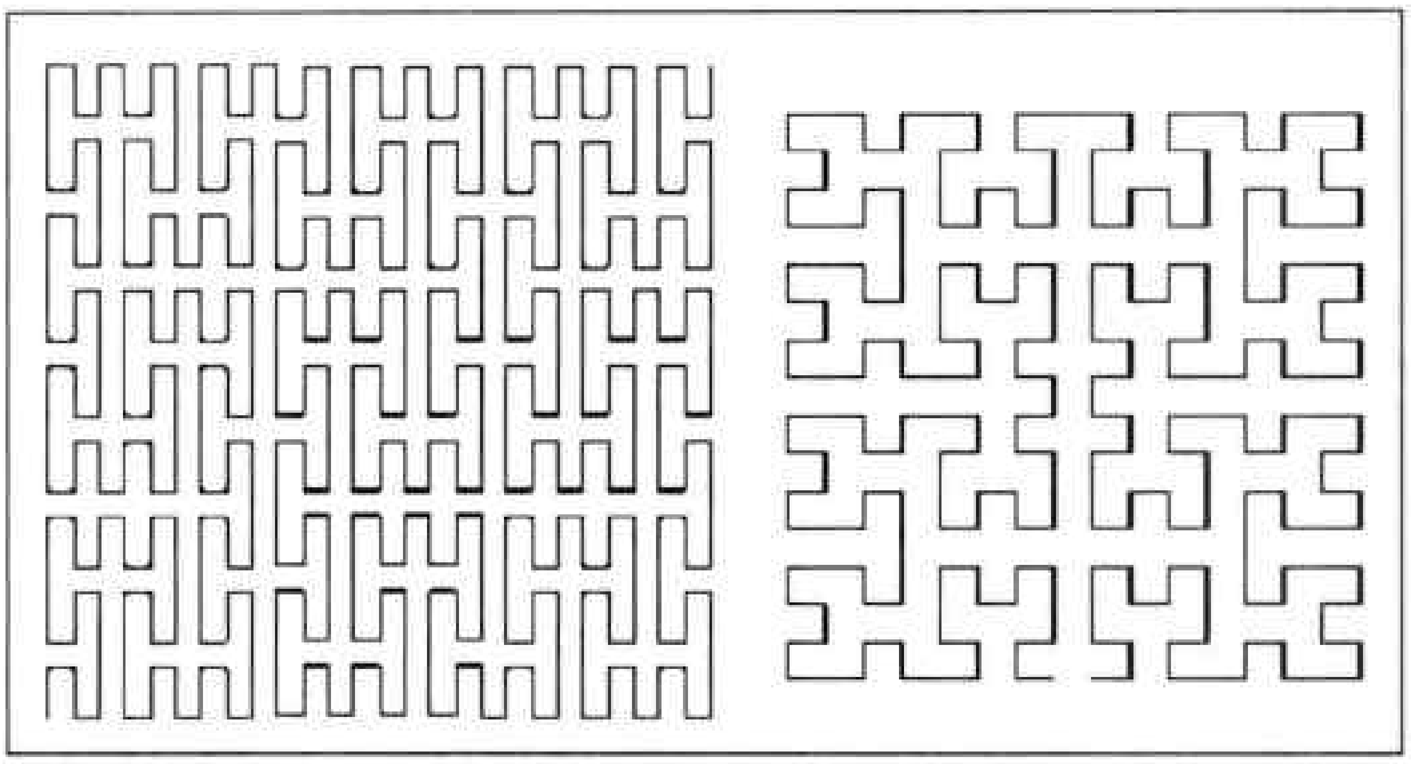
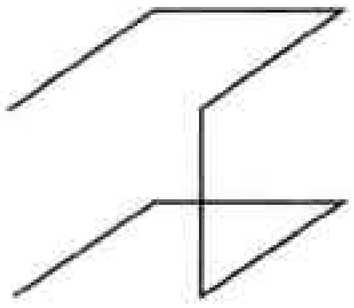


图16.13 皮亚诺曲线（左）与摩尔曲线（右）

希尔伯特曲线可扩展至任意矩形，也可扩展至3维或更高维度的空间中。下面画出了3维希尔伯特曲线的基本构建单元。这条线穿越了8个点，其轮廓是个2×2×2的立方体。[Sagan]一书讲解了上述曲线，此外还讨论了许多其他的空间填充曲线。



16.7 应用

在图像处理领域中，希尔伯特曲线可用来执行压缩、半色调（halftone）[⊖]、文本分析等操作[L & S]，另外也可以提升计算机执行光线追踪算法的效率。光线追踪（ray tracing）是一种图形渲染技术[⊜]。扫描场景时，一般都会按照普通的光栅扫描顺序（raster scan line order，由左到右，然后从上至下扫描屏幕），用横穿场景的光线来投射

⊖ 一种图像复制技术，用大小或频率不同的墨点（半色调网点）来模拟明暗变化。详情参见：https://en.wikipedia.org/wiki/Halftone。——译者注

⊜ 又叫光线跟踪，通过跟踪与光学表面发生互动作用的光线而得到光线途经路径的模型。详情参见：http://zh.wikipedia.org/wiki/光线跟踪。——译者注

物体。如果光线碰到了模拟场景数据库中的物体，那么该物体在此点的颜色及其他属性就能够确定了，可根据这一扫描结果来照亮光线发射方向上的那个像素。（这种说法过于简化，不过用在此处足够了。）这样做会出现一个问题：数据库通常很大，而扫描线可能会穿越多个物体，于是每个物体中的数据必须频繁地调入或调出主内存页面。如果逐行扫描的话，那么在这一根扫描线所碰到的物体中，有很多都是上一根扫描线已经扫描过的，于是又要把那些物体再次调入页面。若扫描操作能局限于某个范围内，则可少执行一些页面调度。比方说，可以先把屏幕的某四分之一彻底扫描完，然后再扫描下一个"象限"。

希尔伯特曲线似乎就具备我们要找的这种"局部性"（locality property）。它会先把某个范围内的四分之一区域完全扫描好，然后再进入下一个四分之一区域，在更大的范围中，它依然按此模式递归扫描，而且在由一个"象限"跨入另一个"象限"时，移动距离不会超过一个单位长度[⊖]。

Douglas Voorhies 先生模拟了通常所用的单向遍历逐行扫描、皮亚诺曲线扫描法以及希尔伯特曲线扫描法这三种方式，研究了每种情况下的分页行为。其方法是，把固定大小的圆形随机散布于屏幕上。如果扫描的路径进入了圆，那么就表示扫描到一个新物体，并将其调入页面。如果扫描线从圆中离开，那么就假定该物体的数据依然会暂时留在内存中，直到扫描线从一个半径为此"物体圆"（"object" circle）二倍的圆中离开时，这些数据才会调出页面。也就是说，如果扫描线在离开某物体没多久之后又回来了，那么不会发生分页操作。他用许多大小不同的圆在 1024×1024 的模拟屏幕上重复了这一实验。

把进入圆形物体并从中离开的过程假定为一次分页操作。那么，要用通常的扫描线来扫描一个直径为 D 像素（并且不太大）的圆，显然需要执行 D 次分页操作，因为每条扫描线都必须进入包围该物体的圆（outer circle），并从中离开。Voorhies 的模拟实验揭示了一个有趣的结果：用皮亚诺曲线来扫描圆，只需 2.7 次分页操作，该次数与圆的直径无关，这一结果也许令人惊讶。如果用希尔伯特曲线来做，那么需要大约 1.4 次，该次数也与圆的直径无关。因此，实验表明，从减少分页操作次数这个角度看，希尔伯特曲线比皮亚诺曲线好，而且远远超越普通的逐行扫描法。（分页操作执行次数与圆直径无关，可能是由外圆直径与圆形物体直径成比例这一人为因素而导致。）

如果在一个 2 维或 3 维的矩形网格（rectangular grid）中有很多相互连接的处理器，那么可用希尔伯特曲线为其指派任务［Cplant］。负责分配处理器的系统软件使用一个线性列表来描述各个处理器，该列表的顺序依据希尔伯特曲线在网格上的走向而定，这与内存分配器（memory allocator）的原理相似。这样分配出来的处理器，在网格上彼此离得近，于是相互通信的效果就比较好。

⊖ 原文为"does not make a long jump"，中文意思是"不会'远跳'（大幅跳跃）"。——译者注

16.8 习题

1. 有一种简单的办法也能覆盖 $n \times n$ 网格，而且点与点之间的大幅跳跃也不多，同时又能穿越每一个点，并保证每个点只经过一次。具体做法是：拿一个 $2n$ 位的变量 s，每轮递增其值，从 s 的第一位开始，每隔一位取一个位元值，这些位元就构成了当前这一步的 x 坐标，从 s 的第二位开始，每隔一位收集一个位元值，这样就能求出 y 坐标。此算法就相当于对 s 执行“外完美理牌”（perfect outer unshuffle）操作，而 x 与 y 就是其结果的左半边和右半边。研究该曲线的“局部性”，并画出 $n=3$ 时的图像。
2. 习题1中的曲线还有个变种：首先把 s 变为格雷码（参见13.1节），然后再按照上一题所用的方式，从中每隔一个位置取一个位元，分别组合成 x 与 y。画出 $n=3$ 时的曲线。这种办法是否改进了“局部性”？
3. 如何仿照习题1的做法生成3维曲线？

第 17 章
浮点数

整数源自上帝，其余皆由人造。

——利奥波德·克罗内克（Leopold Kronecker，1823—1891），德国数学家、逻辑学家

用整数算术与逻辑指令来操作浮点数通常令人手忙脚乱。若用“IEEE 浮点数运算标准”（IEEE Standard for Floating-Point Arithmetic，IEEE Std. 754—2008）来做，则更是如此。此标准通常称为“IEEE 运算”。其中有“NaN”（not a number，非数）和“无穷”（infinity）等概念，而几乎所有操作都可能产生此类特殊值。还有“正 0”（plus zero）和“负 0”（minus zero）这一说法，标准规定这两数必须相等。除了大于、等于、小于这 3 种比较操作之外，还有第 4 种比较操作，名为“unordered”，用以判断其两个操作数是否“无法比较大小”。小数的最高有效位不会出现在“正规形式”（normal）的浮点数里，然而在“非正规形式的”（subnormal）浮点数中则含有此位。在浮点数中，小数部分本身要能表明其正负（signed-true form），指数部分则要由实际指数加上某个偏移量而得，与此相对，整数几乎都使用“二补码”来表示。上述种种做法自然有其原因，但是，这套规则却导致程序里到处都是测试和分支语句，而且很难编出高效实现的代码。

本章假定读者已经熟悉 IEEE 标准了，书中只会简要概述其内容。

17.1 IEEE 格式

2008 年颁布的标准包含 3 种二进制格式和两种十进制格式，我们只关注二进制“单精度”（single）与“双精度”（double）浮点数格式（前者 32 位，后者 64 位）。下面列出这两种格式。

单精度浮点数格式

s	e	f
1	8	23

双精度浮点数格式

s	e	f
1	11	52

如果是正数，符号位 s 就是 0，负数则为 1。在偏移后的指数 e 与小数 f 中，左侧的位元权重高。下面两张表格列出了浮点数编码所对应的具体数值。

单精度浮点数格式

e	f	数值
0	0	± 0
0	$\neq 0$	$\pm 2^{-126}(0, f)$
0至254	—	$\pm 2^{e-127}(1, f)$
255	0	$\pm\infty$
255	$\neq 0$	NaN

双精度浮点数格式

e	f	数值
0	0	± 0
0	$\neq 0$	$\pm 2^{-1022}(0, f)$
1至2046	—	$\pm 2^{e-1023}(1, f)$
2047	0	$\pm\infty$
2047	$\neq 0$	NaN

举个例子，考虑如何把 π 编码为单精度浮点数格式。用二进制表示 [Knu1]：

$\pi \approx$ 11.0010 0100 0011 1111 0110 1010 1000 1000 1000 0101 1010 0011 0000 10...

根据上表第3行可知，此数在“正规格式”数字的范围内。在 π 的浮点数表示中，权重最高的1不用写，因为正规格式的编码不含这个前导1。为了令二进制小数点处于正确位置，指数 e 减去127之后应该是1，因此，$e=128$。于是，π 用浮点数表示出来就是：

```
0 1000 0000 10010010000111111011011
```

或者写成十六进制：

```
40490FDB
```

小数部分已经舍入为距其最近且可以表示的值。

符合 $1 \leqslant e \leqslant 254$ 的数叫做“正规数”（normal number）。这些数已经“正规化了”（normalized），也就是说，其最高有效位是1，所以无需存储这一位。值非0但 $e=0$ 的数叫做“非正规数”（subnormal number，或简称 subnormal）。这些数的最高有效位要存入编码中。此存储方案叫做“逐级下溢”（gradual underflow）。表17.1列出了浮点数在数个取值范围内的某些极值。表格中的“最大整数”是指所有绝对值小于等于该值的数都可以精确地表示出来，而其下一个值则只能舍入之后再表示。

对于单精度格式的正规数来说，最后一位所表示的精度（unit in the last position，简称 ulp）[⊖] 在 $1/2^{24}$ 至 $1/2^{23}$ 之间（约为 5.96×10^{-8} 至 1.19×10^{-7}），对于双精度格式的正规数来说，最后一位所表示的精度在 $1/2^{53}$ 至 $1/2^{52}$ 之间（约为 1.11×10^{-16} 至 2.22×10^{-16}）。如果舍入模式为“向最近值舍入”，那么最大的“相对误差”（relative error）[⊜] 就是上述值的一半。

用单精度浮点数可以精确表示的整数范围是 -2^{24} 至 $+2^{24}$（$-16\ 777\ 216$ 至 $+16\ 777\ 216$），用双精度可以表示的范围是 -2^{53} 至 $+2^{53}$（$-9\ 007\ 199\ 254\ 740\ 992$ 至 $+9\ 007\ 199\ 254\ 740\ 992$）。当然，某些位于该范围之外的整数，（例如值更大的2的幂）也可以精确表示出来，上面两个范围描述的是其中所有整数都能精确表示的最大范围。

⊖ 这一概念的详情可参考：https://en.wikipedia.org/wiki/Unit_in_the_last_place。——译者注

⊜ 按照经典定义，相对误差应指实际值与计算值之差的绝对值占实际绝对值的比例，然而作者在此书中所说的“相对误差”一词有时指计算值与实际值之差。——译者注

表 17.1　极值表

	单精度		
	十六进制	精确值	近似值
最小的非正规数	0000 0001	2^{-149}	1.401×10^{-45}
最大的非正规数	007F FFFF	$2^{-126}\ (1-2^{-23})$	1.175×10^{-38}
最小的正规数	0080 0000	2^{-126}	1.175×10^{-38}
1.0	3F80 0000	1	1
最大整数	4B80 0000	2^{24}	1.677×10^{7}
最大正规数	7F7F FFFF	$2^{128}\ (1-2^{-24})$	3.403×10^{38}
∞	7F80 0000	∞	∞
	双精度		
最小的非正规数	0...0001	2^{-1074}	4.941×10^{-324}
最大的非正规数	000F...F	$2^{-1022}\ (1-2^{-52})$	2.225×10^{-308}
最小的正规数	0010...0	2^{-1022}	2.225×10^{-308}
1.0	3FF0...0	1	1
最大整数	4340...0	2^{53}	9.007×10^{15}
最大正规数	7FFF...F	$2^{1024}\ (1-2^{-53})$	1.798×10^{308}
∞	7FF0...0	∞	∞

我们可能想用乘以除数的倒数这一方式来计算除法。只有当除数的倒数可以准确表示成浮点数时，计算结果才会完全精确（“完全精确”一词是从 IEEE 浮点数的精确标准来说的）。若 2 的幂在 2^{-127} 至 2^{127} 之间，则可准确表示为单精度浮点数，若在 2^{-1023} 至 2^{1023} 之间，则可准确表示为双精度浮点数。2^{-127} 和 2^{-1023} 是非正规数，如果计算机无法高效操作非正规数，那么应尽量避开这些值。

17.2　整数与浮点数互化

表 17.2 列出了在 IEEE 浮点数格式与整数之间互相转换时所用到的公式。这些算法既简明又高效，然而，并非所有输入值都能得出正确结果。表中列出了能够求得精确结果的输入值范围。当输入值为±0.0 或相关范围内的非正规数时，总能得出正确结果。大部分公式在输入值为 NaN 或“无穷”时都无法算出合理结果。在某些应用场景中，可以直接使用下列公式，也可以将其放在程序库（library routine）中，以便快速执行常见情况下的互化操作。

“转换类型”这一栏指明了待转换数与转换结果的类型，同时也标明了计算结果的舍入模式：n 表示向最近的“偶数”取整（round for nearest even）[⊖]，d 表示向下取整，u 表示向上取整，z 表示向 0 取整。R 一栏的意思是：为了令公式能算出正确结果，计算机

⊖ 这里“偶数”一词的含义与自然数不同，故加引号，详情参见：https://en.wikipedia.org/wiki/IEEE_floating_point#Roundings_to_nearest。——译者注

必须运行在何种舍入模式下。（在 Intel IA-32 等架构[⊖]的计算机中，舍入模式可以在指令中指定，而不用修改“模式”寄存器。）

“double”是指 IEEE 规范下的双精度浮点数，占 64 位。“float”是指 IEEE 规范下的单精度浮点数，占 32 位。

“ulp”表示最后一位的精度。例如，1.0-ulp 表示一个 IEEE 格式的浮点数，其值接近 1.0 然而小于 1.0，差不多是 0.99 999…。“int 64”表示（二补码形式的）64 位带符号整数，“int32”表示 32 位带符号整数。“uint64”及“uint32”含义与前述类似，不过都是无符号数。

函数 low32（x）提取出 x 的低权重 32 位。

$\overset{d}{+}$与$\overset{s}{+}$操作符分别表示双精度与单精度浮点数加法。与之类似，$\overset{d}{-}$与$\overset{s}{-}$则分别表示双精度与单精度浮点数减法。

表 17.2 浮点数与整数相互转换规则

转换类型	R	公式	输入值范围	注释
Double 至 int64，n	n	$(x\overset{d}{+}c_{521})-c_{521}$	-2^{51} 至 $2^{51}+0.5$	1
Double 至 int64，d	d	$(x\overset{d}{+}c_{521})-c_{521}$	$-2^{51}-0.5$ 至 $2^{51}+0.5$	1
Double 至 int64，u	u	$(x\overset{d}{+}c_{521})-c_{521}$	-2^{51} 至 $2^{51}+1$	1
double 至 int64，z	d 或 z	if ($x\geqslant 0.0$) $(x\overset{d}{+}c_{52})-c_{52}$ else $c_{52}-(c_{52}\overset{d}{-}x)$	-2^{52} 至 2^{52}	1
Double 至 uint64，n	n	$(x\overset{d}{+}c_{52})-c_{52}$	-0.25 至 2^{52}	1
Double 至 uint64，d	d	$(x\overset{d}{+}c_{52})-c_{52}$	0 至 2^{52}	1
Double 至 uint64，u	u	$(x\overset{d}{+}c_{52})-c_{52}$	$-0.5+\text{ulp}$ 至 $2^{52}+1$	1
Double 至 int32 或 uint32，n	n	low32 $(x\overset{d}{+}c_{521})$	$-2^{31}-0.5$ 至 $2^{31}-0.5-\text{ulp}$，或 -0.5 至 $2^{32}-0.5-\text{ulp}$	1
Double 至 int32 或 uint32，d	d	low32 $(x\overset{d}{+}c_{521})$	-2^{31} 至 $2^{31}-\text{ulp}$，或 0 至 $2^{32}-\text{ulp}$	1
Double 至 int32 或 uint32，u	u	low32 $(x\overset{d}{+}c_{521})$	$-2^{31}-1+\text{ulp}$ 至 $2^{31}-1$，或 $-1+\text{ulp}$ 至 $2^{32}-1$	1
Double 至 int32 或 uint32，z	d 或 z	if ($x\geqslant$**0.0**)low32$(x\overset{d}{+}c_{521})$ else $-$low32$(c_{521}\overset{d}{-}x)$	$-2^{31}-1+\text{ulp}$ 至 $2^{31}-\text{ulp}$，或 $-1+\text{ulp}$ 至 $2^{32}-\text{ulp}$	

[⊖] 英特尔 32 位元架构（Intel Architecture，32-bit，缩写为 IA-32），常称为 i386、x86-32、x86，是英特尔公司于 1985 年推出的指令集架构，为 x86 架构的延伸版本。详情参见：http://zh.wikipedia.org/wiki/IA-32。——译者注

（续）

转换类型	R	公式	输入值范围	注释
Float 至 int32，n	n	$(x \overset{s}{+} + c_{231} - c_{231})$	-2^{22} 至 $2^{22}+0.5$	
Float 至 int32，d	d	$(x \overset{s}{+} c_{231}) - c_{231}$	$-2^{22}-0.5$ 至 $2^{22}+0.5$	
Float 至 int32，u	u	$(x \overset{s}{+} c_{231}) - c_{231}$	-2^{22} 至 $2^{22}+1$	
Float 至 int32，z	d 或 z	if $(x \geqslant 0.0)$ $(x \overset{s}{+} c_{23}) - c_{23}$ else $c_{23} - (c_{23} \overset{s}{-} x)$	-2^{23} 至 2^{23}	
Float 至 uint32，n	n	$(x \overset{s}{+} c_{23}) - c_{23}$	-0.25 至 2^{23}	
Float 至 uint32，d	d	$(x \overset{s}{+} c_{23}) - c_{23}$	0 至 2^{23}	
Float 至 uint32，u	u	$(x \overset{s}{+} c_{23}) - c_{23}$	$-0.5+\text{ulp}$ 至 $2^{23}+1$	
将双精度浮点数舍入为最近整数	n	$(x \overset{d}{+} c_{521}) \overset{d}{-} c_{521}$	-2^{51} 至 $2^{51}+0.5$	1
将非负双精度浮点数舍入为最近整数	n	$(x \overset{d}{+} c_{52}) \overset{d}{-} c_{52}$	-0.25 至 2^{52}	1，3
将单精度浮点数舍入为最近整数	n	$(x \overset{s}{+} c_{231}) \overset{s}{-} c_{231})$	-2^{22} 至 $2^{22}+0.5$	
将非负单精度浮点数舍入为最近整数	n	$(x \overset{s}{+} c_{23}) \overset{s}{-} c_{23}$	-0.25 至 2^{23}	2，3
Int64 至 double	—	$(x + c_{521}) \overset{d}{-} c_{521}$	-2^{51} 至 2^{51}	4
Uint64 至 double	—	$-(x + c_{52} \overset{d}{-} c_{52})$	0 至 $2^{52}-1$	4
Int32 至 float	—	$-(x + c_{231}) \overset{s}{-} c_{231}$	-2^{22} 至 2^{22}	
Uint32 至 float	—	$-(x + c_{23}) \overset{s}{-} c_{23}$	0 至 2^{23}	

常量：

c_{521} = 0x4338 0000 0000 0000 $= 2^{52}+2^{51}$

c_{52} = 0x4330 0000 0000 0000 $= 2^{52}$

c_{231} = 0x4B40 0000 $= 2^{23}+2^{22}$

c_{23} = 0x4B00 0000 $= 2^{23}$

注释：

1. 浮点数操作必须按 IEEE 双精度（53 位精度）模式执行，精确度不能比这个高。大部分 Intel 架构的计算机在默认情况下都不会依此模式运算。在此类计算机上运算时，需要把精确度模式（FPU 控制字的 PC 字段[⊖]）设置为“双精度”（double-precision）。
2. 浮点数操作必须按 IEEE 单精度（24 位精度）模式执行，精确度不能比这个高。大部分 Intel 架构的计算机在默认情况下都不会依此模式运算。在此类计算机上运算时，需要把精确度模式（FPU 控制字的 PC 字段）设置为“单精度”（single-precision）。
3. “非负”（nonnegative）意思是：值为 −0.0 或者大于等于 0.0。
4. 若想把 32 位带符号或无符号整数转为双精度浮点数，可以给 32 位整数左方填充符号位或 0，使之变成 64 位，然后再套用相关公式。

⊖ 详情可参考：http://www.efg2.com/Lab/Library/Delphi/MathFunctions/FPUControlWord.Txt。——译者注

似乎有些奇怪的是：在把双精度浮点数转为（任意大小的）整数时，为了求出正确结果，都必须把大多数Intel架构计算机的精度模式降至53位，然而，若是把单精度浮点数转为整数，则未必要降低精度，采用计算机默认的“扩展精度”模式（extended-precision mode，即64位精度）也能算出正确结果。其原因在于：在按照双精度加法规则与常数相加时，小数部分会向右移位，移动距离也许多达52位，这将把一个二进制位移出64位范围，从而致使该位所表示的数值丢失。因此，在计算过程中有两次舍入，一次是舍入为64位，一次是舍入为53位。与此相反，按单精度加法规则与常数相加时，移位量最多只有23位。如此小的移位量不可能把某个位元移出64位边界，因而只会发生1次舍入操作。在Intel架构计算机的全部三种精度模式下，单精度浮点数都能正确转换为整数。

在使用“扩展精度”模式的Intel架构计算机中，无需改变精度模式就能把双精度浮点数转为64位带符号或无符号整数，只是所用常数不同，而且要多做一次浮点运算。计算方法是：

$$((\boldsymbol{x} \overset{e}{+} \boldsymbol{c}_1) \overset{e}{-} \boldsymbol{c}_2) - \boldsymbol{c}_3$$

其中$\overset{e}{+}$与$\overset{e}{-}$分别表示“扩展精度”的加法与减法。（加法操作的结果必须留在80位寄存器中，以供扩展精度减法操作使用。）

把双精度浮点数转为64位带符号整数所用的常量是：

$\boldsymbol{c}_1 = \text{0x43E0 0300 0000 0000} = 2^{63} + 2^{52} + 2^{51}$
$\boldsymbol{c}_2 = \text{0x43E0 0000 0000 0000} = 2^{63}$
$\boldsymbol{c}_3 = \text{0x4338 0000 0000 0000} = 2^{52} + 2^{51}$

把双精度浮点数转为64位无符号整数所用的常量是：

$\boldsymbol{c}_1 = \text{0x43E0 0200 0000 0000} = 2^{63} + 2^{52}$
$\boldsymbol{c}_2 = \text{0x43E0 0000 0000 0000} = 2^{63}$
$\boldsymbol{c}_3 = \text{0x4330 0000 0000 0000} = 2^{52}$

如果使用上述常量，那么就可以把表17.2中加了注释1的那些公式改写为与之类似且依然包含转换与舍入操作的表达式了。改写后的表达式的适用范围与表中对应公式的输入范围接近。

然而，对于“将双精度浮点数舍入为最近整数”（round double to nearest）[⊖] 这一操作来说，如果先减后加，也就是用下式计算：

$$((\boldsymbol{x} \overset{e}{-} \boldsymbol{c}_1) \overset{e}{+} \boldsymbol{c}_2) + \boldsymbol{c}_3$$

（采用上述第一套常量），那么能算出正确结果的输入值范围就变成了$-2^{51}-0.5$至无穷，然而不能输入NaN。

⊖ 对应表17.2中第19行。——译者注

17.3 使用整数操作比较浮点数大小

IEEE 编码格式有个特点：只要浮点数不是 NaN，就可将其视为带符号整数，并适当排序。

唯有先假设运算结果不会出现“无法比较大小”（unordered）这种情况，然后方能以整数操作比较浮点数。按照 IEEE 754 标准，若比较操作的两操作数有一个为 NaN，或两者皆为 NaN，则结果就是“无法比较大小”。下列方法将 NaN 当成一个其量值比无穷还大的数。

如果认为 −0.0 “严格小于” +0.0 的话（这不合 IEEE 754 标准），那么下列比较操作还能变得再简单些。若上述关系成立，则可按下列算式比较浮点数大小，其中 $\overset{f}{<}$、$\overset{f}{\leqslant}$、$\overset{f}{=}$ 表示浮点数比较操作，≈符号表示该公式无法完全正确处理 ±0.0 问题。这些比较操作与 IEEE 754—2008 标准规定的“全序”（total-ordering）谓词[1]相符。

$$a\overset{f}{=}b\approx(a=b)$$

$$a\overset{f}{<}b\approx(a\geqslant 0\ \&\ a<b)\mid(a<0\ \&\ a\overset{u}{>}b)$$

$$a\overset{f}{\leqslant}b\approx(a\geqslant 0\ \&\ a\leqslant b)\mid(a<0\ \&\ a\overset{u}{\geqslant}b)$$

如果一定要令 −0.0 等于 +0.0 的话，那么似乎没有太巧妙的算法，不过可以考虑下列公式，这些式子都不难由上述算式推出。

$$\begin{aligned}a\overset{f}{=}b&\equiv(a=b)\mid(-a=a\ \&\ -b)=b\\&\equiv(a=b)\mid((a\mid b)=\mathrm{0x8000\ 000})\\&\equiv(a=b)\mid(((a\mid b)\ \&\ \mathrm{0x7FFFFFF})=0)\end{aligned}$$

$$a\overset{f}{<}b\equiv((a\geqslant 0\ \&\ a<b)\mid(a<0\ \&\ a\overset{u}{>}b))\ \&\ ((a\mid b)\neq\mathrm{0x8000\ 000})$$

$$a\overset{f}{\leqslant}b\equiv((a\geqslant 0\ \&\ a\leqslant b)\mid(a<0\ \&\ a\overset{u}{\geqslant}b))\ ((a\mid b)\neq\mathrm{0x8000\ 000})$$

某些情况下，如果先变换操作数，然后再执行一条“定点比较指令”（fixed-point comparison instruction）[2]，那么效率可能会更好。例如，在排序 n 个数时，只需将其中每个数变换一次，然后再按整数方式排序即可，然而比较操作至少要执行 $n\log_2 n$ 次（此处假设尽量选用比较操作次数少的排序算法）。

表 17.3 列出了 4 种变换方式。若规定 −0.0 等于 +0.0，则用左栏；若规定 −0.0 小于 +0.0，则用右栏。不论哪种情况，变换操作都不会改变比较操作符的方向。变量 n 是

---

[1] “全序”一词的含义可参见：https：//zh.wikipedia.org/wiki/全序关系，“全序谓词”的含义可参见：https：//en.wikipedia.org/wiki/IEEE_floating_point#Total-ordering_predicate。——译者注

[2] 在此语境中，可理解为带符号比较操作与无符号比较操作的合称。——译者注

带符号数，t 是无符号数，c 带不带符号均可。

表 17.3 用整数比较操作来比较浮点数之前所做的预处理

−0.0=+0.0（合乎 **IEEE** 标准）	**−0.0<+0.0**（不合 **IEEE** 标准）
if (n>=0) n= n+0x80000000; else n = −n; 执行带符号比较操作.	if (n>=0) n= n+0x80000000; else n = −n; 执行带符号比较操作.

c = 0x7FFFFFFF;
if (n < 0) n = (n^c) +1;
执行带符号比较操作.
	c = 0x7FFFFFFF;
	if (n < 0) n = n^c;
	执行带符号比较操作.

c = 0x8000000;
if (n < 0) n = c − n;
执行带符号比较操作.
	c=0x7FFFFFFF;
	if (n < 0) n=c−n;
	执行带符号比较操作.

t = n ≫ 31; n = (n^ (t ≫ 1))−t; 执行带符号比较操作.	t= (unsigned) (n ≫ 30) ≫1; n = n^t; 执行带符号比较操作.

最后一行列出了无分支代码，如果用本书假定的“基本 RISC 指令”来实现，那么左栏所列算法需要 4 条指令，右栏需要 3 条（待比较的两个操作数都必须先由这 4 条或 3 条指令变换，然后方可比较）。

17.4 估算平方根倒数

在 21 世纪初，编程圈中流传一段奇妙的程序，能估算出 IEEE 单精度浮点数的平方根倒数（reciprocal square root）。这段例程可用于图形处理，例如，要把某向量规格化，可以给其 x、y、z 分量均乘以 $1/\sqrt{x^2+y^2+z^2}$。图 17.1 列出了此函数的 C 语言代码［Taro］。

对于规范的单精度浮点数来说，计算结果的相对误差在 0 与 −0.00 176 之间（也就是说，结果可能低于实际值）。如果参数是 NaN，那么结果也是 NaN，这合乎 IEEE 标准。然而，在参数为 $\pm\infty$、负数或 −0 时，无法算出合理结果。当参数为 +0 或值为正的非正规数时，结果会是个比较大的数（大于 9×10^{18}），此值虽不合预期，但在某些应用场景中仍可使用。

如果在运用牛顿法的那一步里把常量 1.5 改为 1.500 890 8，那么就能减少相对误差的量值，此时误差在 ±0.000 892 之间。

还有一种优化办法，就是把乘以 0.5 这个操作改为将 x 的指数减 1。也就是说，用下列代码替换 xhalf 的定义。

```
union {int ihalf; float xhalf;};
ihalf = ix - 0x00800000;
```

然而，当 x 为小于约 2.34×10^{-38}的正规数时，函数算出的结果不够精确（尽管要比 6×10^{18}大），当 x 为非正规数时，结果是 NaN。当 x=0 时，结果为$\pm\infty$（这是对的）。

其中牛顿法那一步就是按照“平方根倒数函数”(reciprocal square root function) 所对应的“牛顿-拉弗森方法”(Newton-Raphson calculation)[㊀]来算的（参见附录 B)。只要反复执行该步骤，即可把相对误差范围缩减为 0～−0.000 004 7。此时的最优常数为 0x5F37 599E。

另一方面，如果把套用牛顿法的那一步删去，那么函数就能快很多，在常数值为 0x5F37 642F 的情况下，相对误差在±0.035 之间。除去加载常数所需的指令外，这种做法只需要两条整数指令。(若是不执行牛顿法，则 xhalf 变量也可删去。)

```
float rsqrt(float x0) {
   union {int ix; float x;};

   x = x0;                       // x can be viewed as int.
   float xhalf = 0.5f*x;
   ix = 0x5f375a82 - (ix >> 1); // Initial guess.
   x = x*(1.5f - xhalf*x*x);     // Newton step.
   return x;
}
```

图 17.1 估算平方根倒数

为了略微讲述其原理，我们假设 $x=2^n(1+f)$，其中 n 是未偏移的 (unbiased) 指数而 f 是小数 ($0\leqslant f<1$)。那么可知：

$$\frac{1}{\sqrt{x}}=2^{-n/2}(1+f)^{-1/2}$$

如果不计小数部分，那么上式意味着偏移后的指数应该由原来的 $127+n$ 变为现在的 $127-n/2$。设 $e=127+n$，则 $127-n/2=127-(e-127)/2=190.5-e/2$。由此可见，似乎把 x 右移一位，然后从 190 里面减去，就可以粗略估算出 ($1/\sqrt{x}$) 的值了。此思路写为 C 代码就是[㊁]：

```
union {int ix; float x;};  // Make ix and x overlap.
...
0x5F000000 - (ix >> 1);    // Refer to x as integer ix.
```

想分析出比 0x5F000000 更好的常量值似乎比较难。分析时必须考虑 4 种情况：其中两种情况是 0 或 1 由指数字段移到小数字段，另外两种情况是执行减法时，是否产生由小数字段向指数字段的借位。[Lomo] 完成了上述分析。本书只简述其内容。

将 IEEE 单精度浮点数 x 的二进制整数格式记为 rep (x)，我们要寻找下列公式中的常数 k：

㊀ 其中的“Raphson”是指约瑟夫·拉弗森 (Joseph Raphson，约 1648—约 1715)，他与牛顿都提出了这种用迭代法求解方程的思路。——译者注

㊁ 这种写法不合 C 语言规范，然而所有编译器几乎都能编译这段代码。

$$\text{rep}(1/\sqrt{x}) \approx k - (\text{rep}(x) \overset{s}{\gg} 1)$$

（移位操作带不带符号均可，因为 x 为负值和 −0.0 的情况已排除。）选几个 x 代入下式，应该就能大概推知 k 的值了：

$$k \approx \text{rep}\ (1/\sqrt{x}) + (\text{rep}(x) \overset{s}{\gg} 1)$$

表 17.4（以十六进制数）列出了试验结果。

看上去 k 似乎是个常数。注意，x 为 1.0 和 4.0 时算出的 k 相同。实际上，只要 x 和 $4x$ 都是正规数，那么算出的 k 就相同。其原因是，在计算 k 所用的公式中，若 x 变为原来的 4 倍，则 rep $(1/\sqrt{x})$ 这一项的指数字段就要比原来少 1，而 $\text{rep}(x) \overset{s}{\gg} 1$项的指数字段则比原来多 1。

更重要的是，如果 x 与 $4x$ 均为正规数，则它们的估算结果与真实值之间的相对误差一样。其道理在于，我们能够证明，若 rsqrt 函数的参数 x 变为原来的 4 倍，那么函数值恰好为原值的一半，无论把牛顿法那一步执行多少次，都是如此。当然了，此时的 $1/\sqrt{x}$ 也是原来的一半。因此，相对误差不变。

表 17.4 试探常量 k 的最佳取值

试探 k 时所用的 x 值	rep (x)	rep $(1/\sqrt{x})$	k
1.0	3F80 0000	3F80 0000	5F40 0000
1.5	3FC0 0000	3F51 05EC	5F31 05EC
2.0	4000 0000	3F35 04F3	5F35 04F3
2.5	4020 0000	3F21 E89B	5F31 E89B
3.0	4040 0000	3F13 CD3A	5F33 CD3A
3.5	4060 0000	3F08 D677	5F38 D677
4.0	4080 0000	3F00 0000	5F40 0000

此规律很重要，因为这意味着如果能（根据某种标准，例如尽量减少相对误差的最大值）找到适用于 1.0 至 4.0 这个 x 取值范围内的最优 k 值，那么该 k 值对所有正规数来说都最优。

明白了这一点之后，找起来就不难了，编写一段程序，对每一个待测试的 k 值来说，在 1.0 到 4.0 之间取一万多个 x 值试验，每次都把 $1/\sqrt{x}$ 的准确值（可以用一个能算出精确结果的程序库来求此值）和估算值相比，统计出用这一 k 值计算时所产生的最大误差。也可以用手算找出最优 k 值，这么做虽然枯燥，但却很有启发，因为一旦找到这个最佳常数，你就会惊奇地发现，一个仅执行两次整数操作且不用查表法的函数，居然能把误差控制在小于±3.5%的范围内。

17.5 前导数位的分布

IBM 于 1964 年推出了 System/360 计算机，数值分析员（numerical analyst）在发现

此计算机的单精度运算会丢失精度后，都很惊骇。早前的 IBM 计算机，如 704-709-7090 系列[⊖]，其字长都是 36 位。用这种字组所表示的单精度浮点数，其符号与指数字段共有 9 位，后面跟着 27 个二进制位，用来表示小数。由于（在这些机型所定义的“正规数”中）小数部分的最高有效位明确包含在浮点数里，所以用此格式所表示的数值，其精度为 27 位。

S/360 的字长为 32 位。对于单精度浮点数来说，IBM 采用 8 位来保存符号和指数字段，后面跟着 24 位小数字段。从 27 位降为 24 位已经很糟了，不过更麻烦的还在后头。为了扩大 7 位指数字段可表示的指数范围，S/360 格式规定：指数字段中的值每加 1，对应的真实数值就变为原来的 16 倍[⊜]。于是，小数部分的底就成了 16，这种格式也叫做“十六进制”浮点数（“hexadecimal” floating-point）。这样的话，前导数位（leading digit）就有可能是 1 至 15 之间的任意值（对应的二进制为 0001 至 1111）。若前导数位为 1，则只有 21 位精度（因为还有 3 个前导 0），这种情况占所有数值的 1/15(6.7%)。

实际情况比这更糟。大量的分析及使用经验表明，前导数位分布得并不均匀。有 25%的十六进制浮点数，其前导数位都是 1，因而，这些数字只能有 21 位精度。

我们来思考一下十进制数的前导数位分布情况。假设有大量使用科学计数法所表示的数字（scientific notation，例如 6.022×10^{23}），其单位可能是长度、体积、质量、速度等。如果这么一大批数字的前导数位都有明确定义的分布函数，那么它必然和所使用的单位无关，也就是说，无论用英寸还是厘米，用磅还是千克，都不影响其分布。于是，给这一系列数分别乘以同一个常量后，其前导数位的分布依然不变。例如，给所有数字都乘以 2 之后，应该可以确定，前导数位为 1 的数（也就是 1.0 至 1.999... 乘以 10 的若干次方），其个数等于前导数位为 2 或 3 的数（2.0 至 3.999... 乘以 10 的若干次方），因为无论长度单位是英寸还是半英寸，重量单位是千克还是半千克等，都不影响分布。

对于这些带单位的数字来说，设 $f(x)$ 是前导数位的概率密度函数（probability density function），其中 $1\leqslant x<10$。则 $f(x)$ 有个属性：

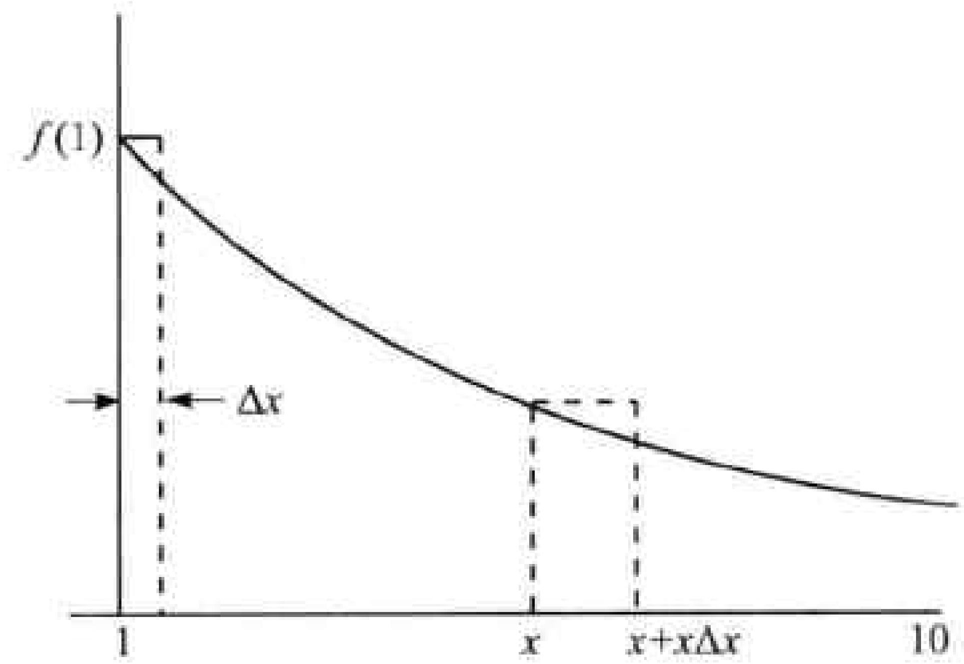
$$\int_a^b f(x)\ \mathrm{d}x$$

就是前导数位在 a 至 b 之间的数在全体数中所占的比例。参考下图，当 x 的增量 Δx 较小时，f 必然满足

$$f(1)\cdot\Delta x=f(x)\cdot x\Delta x$$

⊖ 这 3 个机型分别始于 1954 年、1958 年、1959 年。详情分别参见：http://en.wikipedia.org/wiki/IBM_704、http://en.wikipedia.org/wiki/IBM_709、http://en.wikipedia.org/wiki/IBM_7090。——译者注

⊜ IBM 的浮点数规范可参阅：http://en.wikipedia.org/wiki/IBM_Floating_Point_Architecture——译者注



这是因为 $f(1) \cdot \Delta x$ 约等于值为 1 至 $1+\Delta x$ 的数在全部数字中所占的比例（忽略乘以 10 的若干次幂那一部分），而 $f(x) \cdot x\,\Delta x$ 则约等于值为 x 至 $x+x\,\Delta x$ 的数在全部数字中所占的比例。由于后者是由前者乘以 x 得来的，所以二者在全部数字中所占的比例必然相等。于是，概率密度分布函数有下面这个简单的倒数关系：

$$f(x)=f(1)/x$$

由于在 $x=1$至 $x=10$ 的区间内曲线下方区域的面积是 1（所有数字的前导数位均在 1.000... 至 9.999... 之间），所以很容易证明：

$$f(1)=1/\ln 10$$

前导数位在 a 与 b 之间的数（$1\leqslant a\leqslant b<10$）占全部数字的比例是：

$$\int_a^b \frac{dx}{x\ln 10}=\left.\frac{\ln x}{\ln 10}\right|_a^b=\frac{\ln b/a}{\ln 10}=\log_{10}\frac{b}{a}$$

因此，在十进制下，前导数位是 1 的数占全部数字的比例是 $\log_{10}(2/1)\approx 0.301\,03$，前导数位是 9 的数占全部数字的比例是 $\log_{10}(10/9)\approx 0.0458$。

对于十六进制的情况，也可以推出，前导数位为 a 至 b 的数占全部数字的比例是 $\log_{16}(b/a)$，其中 $1\leqslant a\leqslant b<16$。因此，前导数位是 1 的数所占比例是 $\log_{16}(2/1)=1/\log_2 16=0.25$。

17.6 杂项数值表

表 17.5 列出了一些可能会用到的数值以及与之对应的 IEEE 格式浮点数。这些数未必都是精确值，它们已经舍入为距其最近且可表示的值了。

表 17.5 杂项数值表

十进制数	单精度格式（十六进制）	双精度格式（十六进制）
$-\infty$	FF80 0000	FFF0 0000 0000 0000
−2.0	C000 0000	C000 0000 0000 0000
−1.0	BF80 0000	BFF0 0000 0000 0000
−0.5	BF00 0000	BFE0 0000 0000 0000
−0.0	8000 0000	8000 0000 0000 0000

（续）

十进制数	单精度格式（十六进制）		双精度格式（十六进制）			
+0.0	0000	0000	0000	0000	0000	0000
值最小且为正的非正规数	0000	0001	0000	0000	0000	0001
值最大的非正规数	007F	FFFF	000F	FFFF	FFFF	FFFF
值最小且为正的正规数	0080	0000	0010	0000	0000	0000
$\pi/180$ (0.01745...)	3C8E	FA35	3F91	DF46	A252	9D39
0.1	3DCC	CCCD	3FB9	9999	9999	999A
$\log_{10}2$(0.3010...)	3E9A	209B	3FD3	4413	509E	79EE
$1/e$(0.3678...)	3EBC	5AB2	3FD7	8B56	362C	EF38
1/ln 10 (0.4342...)	3EDE	5BD9	3FDB	CB7B	1526	E50E
0.5	3F00	0000	3FE0	0000	0000	0000
ln 2(0.6931...)	3F31	7218	3FE6	2E42	FEFA	39EF
$1/\sqrt{2}$(0.7071...)	3F35	04F3	3FE6	A09E	667F	3BCD
1/ln 3(0.9102...)	3F69	0570	3FED	20AE	03BC	C153
1.0	3F80	0000	3FF0	0000	0000	0000
ln 3(1.0986...)	3FBC	9F54	3FF1	93EA	7AAD	030B
$\sqrt{2}$(1.414...)	3FB5	04F3	3FF6	A09E	667F	3BCD
1/ln 2(1.442...)	3FB8	AA3B	3FF7	1547	652B	82FE
$\sqrt{3}$(1.732...)	3FDD	B3D7	3FFB	B67A	E858	4CAA
2.0	4000	0000	4000	0000	0000	0000
ln10(2.302...)	4013	5D8E	4002	6BB1	BBB5	5516
e(2.718...)	402D	F854	4005	BF0A	8B14	5769
3.0	4040	0000	4008	0000	0000	0000
π(3.141...)	4049	0FDB	4009	21FB	5444	2D18
$\sqrt{10}$(3.162...)	404A	62C2	4009	4C58	3ADA	5B53
$\log_2 10$(3.321...)	4054	9A17	400A	934F	0979	A371
4.0	4080	0000	4010	0000	0000	0000
5.0	40A0	0000	4014	0000	0000	0000
6.0	40C0	0000	4018	0000	0000	0000
2π(6.283...)	40C9	0FDB	4019	21FB	5444	2D18
7.0	40E0	0000	401C	0000	0000	0000
8.0	4100	0000	4020	0000	0000	0000
9.0	4110	0000	4022	0000	0000	0000
10.0	4120	0000	4024	0000	0000	0000
11.0	4130	0000	4026	0000	0000	0000
12.0	4140	0000	4028	0000	0000	0000
13.0	4150	0000	402A	0000	0000	0000
14.0	4160	0000	402C	0000	0000	0000
15.0	4170	0000	402E	0000	0000	0000
16.0	4180	0000	4030	0000	0000	0000
$180/\pi$(57.295...)	4265	2EE1	404C	A5DC	1A36	C1F8
$2^{23}-1$	4AFF	FFFE	415F	FFFF	C000	0000
2^{23}	4B00	0000	4160	0000	0000	0000
$2^{24}-1$	4B7F	FFFF	416F	FFFF	E000	0000
2^{24}	4B80	0000	4170	0000	0000	0000

（续）

十进制数	单精度格式（十六进制）		双精度格式（十六进制）			
$2^{31}-1$	4F00	0000	41DF	FFFF	FFC0	0000
2^{31}	4F00	0000	41E0	0000	0000	0000
$2^{32}-1$	4F80	0000	41EF	FFFF	FFE0	0000
2^{32}	4F80	0000	41F0	0000	0000	0000
2^{52}	5980	0000	4330	0000	0000	0000
2^{63}	5F00	0000	43E0	0000	0000	0000
2^{64}	5F80	0000	43F0	0000	0000	0000
最大的正规数	7F7F	FFFF	7FEF	FFFF	FFFF	FFFF
∞	7F80	0000	7FF0	0000	0000	0000
“最小的” SNaN	7F80	0001	7FF0	0000	0000	0001
“最大的” SNaN	7FBF	FFFF	7FF7	FFFF	FFFF	FFFF
“最小的” QNaN	7FC0	0000	7FF8	0000	0000	0000
“最大的” QNaN	7FFF	FFFF	7FFF	FFFF	FFFF	FFFF

IEEE 754 没有规定如何区分“signal NaN”（信号 NaN，简称 SNaN）和“quiet NaN”（静默 NaN，QNaN）[㊀]。表 17.5 所采用的这种表示方法使用于 PowerPC、AMD 29050[㊁]、Intel x86 及 I860[㊂]、SPARC、ARM 系列等 CPU 中，其规则是，小数部分最高有效位若是 0，则为 SNaN，若是 1，则为 QNaN。还有一些计算机（尤其是那些老机型）使用相反的约定（0 表示 QNaN，1 表示 SNaN）。

17.7 习题

1. 什么样的数字其单精度格式与双精度格式相同？也就是说，其双精度格式只不过是在单精度格式后面加了 32 个 0 而已。
2. 书中曾提到一段估算平方根倒数的程序，那么请问，有没有类似程序能估算平方根？
3. 给定一个非负的正规数，有没有相似程序能估算其立方根？
4. 有没有与书中所讲方法类似，且能适用于双精度浮点数的平方根倒数估算程序？假定这段程序在 64 位计算机中执行，或者说此计算机支持“long long”数据类型（64 位整数）。

㊀ 二者的作用可参见：http://en.wikipedia.org/wiki/NaN。——译者注

㊁ 由 AMD 研发的一种 32 位 RISC 微处理器，属于 Am29000 系列，详情参见：https://en.wikipedia.org/wiki/AMD_Am29000。——译者注

㊂ Intel 于 1989 年推出的 RISC 微处理器。详情参见：https://en.wikipedia.org/wiki/Intel_i860。——译者注

第18章 素数公式

18.1 简介

与很多年轻学生一样，笔者当年也曾迷恋素数，并且想找到一个素数公式。当时并不清楚在这样一个“公式”中应该包含何种运算，也不确定要找的函数究竟是什么：是根据 n 值生成第 n 个素数，还是根据前面的素数找出新的素数，抑或只要生成的数中有素数就行，又或者是别的东西。尽管不太明确，但笔者还是要对此问题目前的研究成果略说一二。学完这部分内容后，就会明白：(a) 确实有公式能生成素数，(b) 这些公式都不是特别理想。

这一话题中的很多内容都和本书主题有关，因为处理这些公式所用的技巧与某些编程技巧相似，只不过前者是在实数范围内运算，而后者执行“计算机算术”。首先来回顾历史上关于素数的几个重要公式。

1640年，费马（Fermat，1601—1665）猜测，下面这个公式总能算出素数：

$$F_n = 2^{2^n} + 1$$

这种形式的数就叫做“费马数”（Fermat number）。当 n 等于0至4时，F_n的确是素数，然而欧拉（Euler，1707—1783）在1732年发现，

$$F_5 = 2^{2^5} + 1 = 641 \cdot 6\ 700\ 417$$

（在讲如何于32位计算机中求除数为常量的除法时曾经提到上述因子。）此后，F. Landry 于1880年证明：

$$F_6 = 2^{2^6} + 1 = 274\ 177 \cdot 67\ 280\ 421\ 310\ 721$$

对于许多稍大一些的 n 值来说，我们已经知道 F_n 肯定是合数了，例如当 n 在7至16之间（含这两个数）时就是如此。在 $n>4$ 的情况下还没找到素数［H & W］。关于这个臆断[⊖]就讲这么多吧。

顺便说一下费马在公式中使用“双指数”（double exponential）的原因。他认为，如果 m 有除了1之外的奇数因子，那么 2^m+1 就是合数。这是因为，若 $m=ab$，而 b 为不等

⊖ 在费马所提出的猜想中，只有这个是已经证伪的。［Wells］

于 1 的奇数，则

$$2^{ab}+1=(2^a+1)(2^{a(b-1)}-2^{a(b-2)}+2^{a(b-3)}-\ldots+1$$

了解到这一点之后，他肯定在想，如果 m 不包含除 1 之外的任何奇数因子（$m=2^n$），那么 2^m+1 会如何呢？在试验了几个 n 值之后，费马觉得，$2^{2^n}+1$ 看上去应该是素数。

“公式”可以是一个多项式，这一点大家自然都认同。莱昂哈德·欧拉于 1772 年找到了一个神奇的多项式。他发现

$$f(n)=n^2+n+41$$

在 n 为 0 至 39 时都是素数。这一范围还可以扩大。因为

$$f(-n)=n^2-n+41=f(n-1)$$

所以当 n 在 1 至 40 之间时，$f(-n)$ 是素数，这也就等于说，当 n 在 -1 至 -40 之间时，$f(n)$ 是素数。于是：

$$f(n-40)=(n-40)^2+(n-40)+41=n^2-79n+1601$$

在 n 位于 0 至 79 之间时，值为素数。（然而上式不够优雅，因为它不是单调函数，而且值会重复，也就是说，当 $n=0$、1、…、79 时，$n^2-79\ n+1601=1601$、1523、1447、43、41、41、43、…、1447、1523、1601。）

尽管有此成果，但是现在已经能够证明：（除了 $f(n)=5$ 这样的常量多项式之外）没有对于每个 n 值都能产生素数的多项式 $f(n)$。实际上，凡是“非平凡”且“带指数的多项式”几乎都会生成无限多个合数。按照［H & W］一书，更准确的说法应该是：

定理 *若 $f(n)=P(n,2^n,3^n,\cdots,k^n)$ 是以 P 的各参数为项的整数系数多项式，且当 $n\to\infty$ 时 $f(n)\to\infty$，则有无限多个 n 值可令 $f(n)$ 为合数。*

因此，像 $n^2\cdot2^n+2n^3+2n+5$ 这样的公式必然会产生无限多个合数。另一方面，此定理没有论断含有诸如 2^{2^n}、n^n、$n!$ 这种项的多项式。

有个公式可根据 n 值求出第 n 个素数，它用到了向下取整函数及下面这个神奇数字：

$$a=0.203\ 005\ 000\ 700\ 011\ 000\ 013\cdots$$

如果用十进制数来表示 a，那么小数点后第 1 位就是第 1 个素数，其后两位就是第 2 个素数，再其后的 3 位则是第 3 个素数，依此类推。因为 $p_n<10^n$，所以总能用 n 个数的数位写下第 n 个素数。此处不证明 $p_n<10^n$ 这个式子，只是告诉大家，当 $n\geqslant2$ 时，在 n 与 $2n$ 之间总会有一个素数，因此，在 n 至 $10n$ 之间当然至少会有 1 个素数，于是就能推出 $p_n<10^n$。计算第 n 个素数的公式是：

$$p_n=\left\lfloor 10^{\frac{n^2+n}{2}}a\right\rfloor-10^n\left\lfloor 10^{\frac{n^2-n}{2}}a\right\rfloor$$

公式推导过程中用到了 $1+2+3+\cdots+n=(n^2+n)/2$ 这个关系式。例如，

$$\begin{aligned}p_3&=\lfloor 10^6a\rfloor-10^3\lfloor 10^3a\rfloor\\&=203\ 005-203\ 000\\&=5\end{aligned}$$

因为需要把结果预先放在 a 里面，所以这一技巧价值不太大。假如能用和素数无关的方式来定义 a，那么这个公式就很有意义了，可惜没人知道这种定义方式。

显然，使用这一技巧也可以写出许多数列的生成公式来，不过，它却引发了我们下面要讲的问题。

18.2 Willans 公式

C. P. Willians 给出了下面这个公式，用于计算第 n 个素数［Will］：

$$p_n = 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{n} \left(\sum_{x=1}^{m} \left\lfloor \cos^2 \pi \frac{(x-1)!+1}{x} \right\rfloor \right)^{-1/n} \right\rfloor$$

该公式由“威尔逊定理”（Wilson's theorem）开始推导，此定理说，当且仅当 $(p-1)! \equiv -1 (\bmod p)$ 时，p 才会是素数或 1。因此，当 x 是素数或 1 时，

$$\frac{(x-1)!+1}{x}$$

为整数，当 x 是合数时，上式为分数。于是，

$$F(x) = \left\lfloor \cos^2 \pi \frac{(x-1)!+1}{x} \right\rfloor = \begin{cases} 1, x \text{ 为素数或 } 1 \\ 0, x \text{ 为合数} \end{cases} \tag{1}$$

如果用 $\pi(m)$ 表示小于等于 m 的素数个数[⊖]，那么

$$\pi(m) = -1 + \sum_{x=1}^{m} F(x) \tag{2}$$

另外还知道 $\pi(p_n)=n$，而且：

$$\text{如果 } m < p_n \text{，则 } \pi(m) < n \text{，}$$
$$\text{如果 } m \geqslant p_n \text{，则 } \pi(m) \geqslant n$$

因此，在 1 至∞范围内，能令 $\pi(m)<n$ 成立的 m 值共有 p_n-1 个。也就是说，

$$p_n = 1 + \sum_{m=1}^{\infty} (\pi(m) < n) \tag{3}$$

该式中，累加操作的被加数是一个“谓词表达式”(其值非 0 即 1)。

由于 π (m) 已经可以根据 (2) 式算出来了，所以，(3) 就成了计算第 n 个素数所用的公式，此式为 n 的函数。但是，该式有两个地方可能无法令人接受：一个是无限累加 (infinite summation)，一个是“谓词表达式”，这不是标准的数学用法。

已经证明，在 $n \geqslant 1$ 时，n 和 $2n$ 之间至少有 1 个素数。因此，小于等于 2^n 的素数至少有 n 个，也就是说，$\pi(2^n) \geqslant n$。因此，当 $m \geqslant 2^n$ 时，$\pi(m)<n$ 这个谓词就是 0，于是 (3) 式的累加上限可改写为 2^n。

⊖ 非常抱歉，刚才那个式子和这个函数名里都出现了字符“π”，两者意义不同（前者为圆周率，后者为小于等于 m 的素数个数），然而这就是标准的记法，应当不会引发混淆。

Willans 用一种相当巧妙的办法替换了谓词表达式。设

$$\mathrm{LT}(x,y)=\left\lfloor \sqrt[y]{\frac{y}{1+x}}\right\rfloor,(\text{其中 } x=0,1,2,\cdots \text{ 而 } y=1,2,\dots)$$

那么可以看出，若 $x<y$，则 $1\leqslant y/(1+x)\leqslant y$，所以此时有 $1\leqslant \sqrt[y]{y/(1+x)}\leqslant\sqrt[y]{y}<2$。而且，若 $x\geqslant y$，则 $0<y/(1+x)<1$，所以此时有 $0\leqslant \sqrt[y]{y/(1+x)}<1$。再运用向下取整函数，于是可得：

$$\mathrm{LT}(x,y)=\begin{cases}1,x<y,\\0,x\geqslant y\end{cases}$$

也就是说，$\mathrm{LT}(x,y)$ 就是 $x<y$ 这个谓词的值（x 与 y 必须在给定范围内取值）。

用此替换等式（3）中的谓词，可将此式其改写为：

$$\begin{aligned}p_n&=1+\sum_{m=1}^{2^n}\mathrm{LT}(\pi(m),n)\\&=1+\sum_{m=1}^{2^n}\left\lfloor\sqrt[n]{\frac{n}{1+\pi(m)}}\right\rfloor\end{aligned}$$

用等式（2）替换上式中的 $\pi(m)$，即可将其改写为与 $F(x)$ 相关的等式，再用等式（1）替换掉其中的 $F(x)$，于是就得出本节开头那个公式了。

18.2.1 Willans 第二公式

Willans 又给出一个公式：

$$p_n=\sum_{m=1}^{2^n}mF(m)\lfloor 2^{-|\pi(m)-n|}\rfloor$$

其中 F 和 π 函数与第一个公式所说的意义相同。于是，如果 m 是素数或 1，那么 $mF(m)$ 就是 m，否则就是 0。在累加操作的被加数中，第 3 个因子实际上就是谓词 $\pi(m)=n$。在累加操作的各项中，有一项是第 n 个素数，其余项全都是 0。例如：

$$\begin{aligned}p_4&=1\cdot1\cdot0+2\cdot1\cdot0+3\cdot1\cdot0+4\cdot0\cdot0+5\cdot1\cdot0+6\cdot0\cdot0+7\cdot1\cdot1\\&\quad+8\cdot0\cdot1+9\cdot0\cdot1+10\cdot0\cdot1+11\cdot1\cdot0+\cdots+16\cdot0\cdot0\\&=7\end{aligned}$$

18.2.2 Willans 第三公式

Willans 继续给出计算第 n 个素数的公式，这次没有使用向下取整或求绝对值等“非解析”（nonanalytic）[⊖] 函数。首先他注意到，当 $x=2$，3，…时，函数

$$\frac{((x-1)!)^2}{x}=\begin{cases}\text{整数}+\dfrac{1}{x},\text{当 } x \text{ 为素数时}\\\text{整数},\text{当 } x \text{ 为合数或 } 1\end{cases}$$

⊖ 这是笔者自己的用词，并非 Willans 所说。

其中第一种情况是由下式推导出来的：

$$\frac{((x-1)!)^2}{x}=\frac{((x-1)!+1)\cdot((x-1)!-1)}{x}+\frac{1}{x}$$

根据威尔逊定理可知，x 为素数时，$(x-1)!+1$ 能为 x 所整除。因此，当 $x\geqslant 2$ 时，"x 为素数"这一谓词的值可由下式算出：

$$H(x)=\frac{\sin^2 \pi\dfrac{((x-1)!)^2}{x}}{\sin^2\dfrac{\pi}{x}}$$

由上式可推出：

$$\pi(m)=\sum_{x=2}^{m}H(x),(\text{其中 } m=2,3,\cdots)$$

不能按照推导前两个公式时所用的方法来转换上式，因为那时用到了向下取整函数，而现在不能用这个函数。Willans 按照下列公式[⊖]来求 $x<y$ 这个谓词，其中 x、y 大于等于 1：

$$\mathrm{LT}(x,y)=\sin\left(\frac{\pi}{2}\cdot 2^e\right),\text{其中}$$

$$e=\prod_{i=0}^{y-1}(x-i)$$

因此，若 $x<y$，则 $e=x(x-1)\cdots(0)(-1)\cdots(x-(y-1))=0$，所以此时有 $\mathrm{LT}(x,y)=\sin(\pi/2)=1$。若 $x\geqslant y$，则连乘操作的因子里不包括 0，于是 $e\geqslant 1$，所以此时有 $\mathrm{LT}(x,y)=\sin((\pi/2)\cdot(\text{偶数}))=0$。

最后，与推导第一个 Willans 公式时相似，我们把 p_n 写成

$$p_n=2+\sum_{m=2}^{2^n}\mathrm{LT}(\pi(m),n)$$

然后全部展开，就可以得出下面这个相当可畏的公式了：

$$p_n=2+\sum_{m=2}^{2^n}\sin\left(\frac{\pi}{2}\cdot 2^{\prod_{i=0}^{n-1}\left(\sum_{x=2}^{m}\frac{\sin^2\pi\frac{((x-1)!)^2}{x}}{\sin^2\frac{\pi}{x}}-i\right)}\right)$$

18.2.3 Willans 第四公式

Willans 给出了一个根据 p_n 来求 p_{n+1} 的公式：

$$p_{n+1}=1+p_n+\sum_{i=1}^{2p_n}\prod_{j=1}^{i}f(p_n+j)$$

其中 $f(x)$ 表示当 $x\geqslant 2$ 时，"x 为合数"这一谓词的值，也就是说：

⊖ 笔者略微简化了 Willans 原来所用的公式。

$$f(x)=\left\lfloor \cos^2\pi\frac{((x-1)!)^2}{x}\right\rfloor$$

如果不想令公式中出现向下取整函数的话，也可以把 $f(x)$ 定义为 $1-H(x)$。

我们用 $p_n=7$ 为例来演示此公式。根据公式，此时可得：

$$\begin{aligned}p_{n+1} &= 1+7+f(8)+f(8)f(9)+f(8)f(9)f(10)\\ &\quad +f(8)f(9)f(10)f(11)+\cdots+f(8)f(9)\cdots f(14)\\ &=1+7+1+1\cdot 1+1\cdot 1\cdot 1+1\cdot 1\cdot 1\cdot 0+\cdots+1\cdot 1\cdot 1\cdot 0\cdot 1\cdot 0\cdot 1\\ &=11\end{aligned}$$

18.3 Wormell 公式

C. P. Wormell 改进了 Willans 公式，他既没使用三角函数，也没使用向下取整函数[Wor]。原则上讲，只需简单编写一个仅执行整数运算的计算机程序，即可求出 Wormell 公式的值。推导过程中没有使用威尔逊定理。首先，当 $x\geqslant 2$ 时，

$$B(x)=\prod_{a=2}^{x}\prod_{b=2}^{x}(x-ab)^2=\begin{cases}\text{正整数},\text{若 } x \text{ 为素数}\\ 0,\text{若 } x \text{ 为合数}\end{cases}$$

因此，小于等于 m 的素数个数可由下式给出：

$$\pi(m)=\sum_{x=2}^{m}\frac{1+(-1)^{2^{B(x)}}}{2}$$

其中累加操作的被加数等于“x 为素数”这一谓词的值。

当 $n\geqslant 1$ 且 $a\geqslant 0$ 时，可以看出：

$$\prod_{r=1}^{n}(1-r+a)^2=\begin{cases}0,a<n,\\ \text{正整数},a\geqslant n\end{cases}$$

再使用一次刚才定义 $\pi(m)$ 时的技巧，把 $a<n$ 这一谓词的值写成：

$$(a<n)=\frac{1-(-1)^{2^{\prod_{r=1}^{n}(1-r+a)^2}}}{2}$$

由于我们已经知道

$$p_n=2+\sum_{m=2}^{2^n}\mathrm{LT}(\pi(m)<n)$$

所以，把累加操作中的常数项提取出来，即可得到：

$$p_n=\frac{3}{2}+2^{n-1}-\frac{1}{2}\sum_{m=2}^{2^n}(-1)^{2^{\prod_{r=1}^{n}\left[1-r+\frac{(m-1)}{2}+\frac{1}{2}\sum_{x=2}^{m}(-1)^{2^{\prod_{a=2}^{x}\prod_{b=2}^{x}(x-ab)^2}}\right]^2}}$$

正如本节开头所说，Wormell 公式没有使用三角函数。然而他也指出，若是根据 $(-1)^n=\cos\pi n$ 这一等式把 -1 的幂展开，那么又会重新出现三角函数。

18.4 用公式来描述其他难解的函数

现在仔细看看 Willans 和 Wormell 的研究成果。我们用下列规则来定义一类函数，这些函数的值都可以用“公式”算出来，于是可称之为“公式函数”（formula function）。规则里的 $\overline{x}$ 是 x_1，x_2，…，x_n的缩写（其中 $n\geqslant 1$）。公式函数的值域是…-2，-1，0，1，2，…这样的整数。

1. 像…-1，0，1，…这样的常量是公式函数。

2. 当 $1\leqslant i\leqslant n$ 时，投影函数 $f(\overline{x})=x_i$ 是公式函数。

3. 如果 x 与 y 都是公式函数，那么表达式 $x+y$、$x-y$、xy 也是公式函数。

4. 公式函数在组合（替换）操作下闭合㊀。也就是说，如果 f 和 g_i 都是公式函数（其中 $i=1$，…，m），那么 $f(g_1(\overline{x}), g_2(\overline{x}), \cdots, g_m(\overline{x}))$ 也是公式函数。

5. 若 a、b、f 均为公式函数，且 $a(\overline{x})\leqslant b(\overline{x})$，则下列有界累加（bounded sum）与有界连乘（bounded product）算式均为公式函数

$$\sum_{i=a(\overline{x})}^{b(\overline{x})}{}' f(i,\overline{x}) \qquad \prod_{i=a(\overline{x})}^{b(\overline{x})}{}' f(i,\overline{x})$$

累加与连乘操作之所以必须有界，是因为只有这样做才能令公式可以计算，换句话说，只要反复代入各参数，并通过有限次的计算，就能求出公式的值。本节稍后将会解释带′符号（prime，角分符号）的累加操作（$\sum'$）及连乘操作（$\prod'$）与通常含义有何不同。

在通过替换操作构造新公式时，我们会依照惯例添加必要的括号。

注意，上述规则里没有包含除法，因为该操作太复杂了，不能轻易当做“公式函数”。即便如此，这个列表也不是最精简的。寻找“公式函数”一词的最简定义应该是件蛮有趣的事，但我们在这里就不详细讨论了。

这里所定义的“公式函数”一词与 [Cut] 一书讲的“初等函数”相仿。然而，那本书用的值域是非负整数（在“递归函数论”（recursive function theory）中一般使用这个值域）。此外，[Cut] 一书要求连加与连乘运算的操作范围是 0 至 $x-1$（x 为变量），而且还允许“空的”（vacuous）操作范围（在这种情况下，累加操作的结果是 0，连乘操作的结果是 1）。

接下来将要告诉大家，公式函数的范围相当广泛，包括数学中常见的大部分函数。但是，不能说所有定义简单且具有初等函数性质的函数都包含在公式函数这一范围内。

与讲述递归函数论的过程相比，本书在推演公式函数时会遇到一点小麻烦，因为变

㊀ “闭合”是个数学用语，如果某个运算的源操作数和运算结果都总是处于同一个范围，那么就可以说此范围在该运算下闭合，或者说，此范围对于该运算封闭。例如两整数相减，结果仍是整数，所以可说整数在减法操作下是闭合的。——译者注

量可以取负值。如果某个 1 次幂表达式的值可能为负，那么只需将其平方，就不会出现负数了。另外，我们还规定了累加与连乘运算的操作范围不得为空，这也会令问题变得稍微复杂一些。

此处所说的“谓词”就是“0/1 值函数”（0/1-valued function），但是递归函数论中的谓词则是“真/假值函数”（true/false-valued function），而且每个谓词都有一个与之相关的“特征函数”（characteristic function），那个特征函数是“0/1 值函数”。同编程语言和计算机算术的常见做法类似，我们把 1 和“真”关联起来，把 0 和“假”关联起来，（“与”和“或”指令就是这样做的），而在逻辑学与递归函数论中，关联方式通常与此相反。

下面列出公式函数：

1. $a^2=aa$、$a^3=aaa$，等等。

2. 谓词 $a=b$：

$$(a=b)=\prod_{j=0}^{(a-b)^2}{}'(1-j)$$

3. $(a\neq b)=1-(a=b)$

4. 谓词 $a\geqslant b$：

$$\begin{aligned}(a\geqslant b)&=\sum_{i=0}^{(a-b)^2}{}'((a-b)=i)\\&=\sum_{i=0}^{(a-b)^2}{}'\prod_{j=0}^{((a-b)-i)^2}{}'(1-j)\end{aligned}$$

刚才说过，按惯例，若累加与连乘的运算范围是“空”，则其值分别为 0 和 1，现在笔者来解释本书为何不遵照这一惯例。假如依照这一惯例，那么就会推出下面这两个刻意卖弄的式子：

$$(a=b)=\sum_{i=0}^{-(a-b)^2}1,(a\geqslant b)=\prod_{i=a}^{b-1}0$$

对下面要列出的函数来说，比较谓词相当关键，所以笔者不想将其构建在这种太过做作的公式之上。

5. $(a>b)=(a\geqslant b+1)$

6. $(a\leqslant b)=(b\geqslant a)$

7. $(a<b)=(b>a)$

8. $|a|=(2(a\geqslant 0)-1)a$

9. $\max(a,b)=(a\geqslant b)(a-b)+b$

10. $\min(a,b)=(a\geqslant b)(b-a)+a$

现在我们来处理累加与连乘运算，以便在操作范围是“空”的时候也能算出符合惯例且有意义的结果来。

11. $\sum_{i=a(\overline{x})}^{b(\overline{x})} f(i,\overline{x}) = (b(\overline{x}) \geqslant a(\overline{x})) \sum_{i=a(\overline{x})}^{\max(a(\overline{x}),b(\overline{x}))}{}' f(i,\overline{x})$

12. $\prod_{i=a(\overline{x})}^{b(\overline{x})} f(i,\overline{x}) = 1 + (b(\overline{x}) \geqslant a(\overline{x}))(-1 + \prod_{i=a(\overline{x})}^{\max(a(\overline{x}),b(\overline{x}))}{}' f(i,\overline{x}))$

从现在起，累加符号（Σ）和连乘符号（Π）右上角就不带 $'$ 了。这样一来，下面定义的函数就是“全函数”了（也就是对全部参数值都有定义的函数）。

13. $n! = \prod_{i=1}^{n} i$

当 $n \leqslant 0$ 时，若按上式运算，则结果是 $n! = 1$。

下面各式中的 P 与 Q 均是谓词。

14. $\neg P(\overline{x}) = 1 - P(\overline{x})$

15. $P(\overline{x}) \& Q(\overline{x}) = P(\overline{x}) Q(\overline{x})$

16. $P(\overline{x}) \mid Q(\overline{x}) = 1 - (1 - P(\overline{x}))(1 - Q(\overline{x}))$

17. $P(\overline{x}) \oplus Q(\overline{x}) = P(\overline{x}) - Q(\overline{x}))^2$

18. “如果 $P(\overline{x})$ 成立，那么值为 $f(\overline{y})$，否则值为 $g(\overline{z})$”这一操作的结果，可用此公式计算：

$$P(\overline{x}) f(\overline{y}) + (1 - P(\overline{x})) g(\overline{z})。$$

19. a^n = 若 $n \geqslant 0$ 则 $\prod_{i=1}^{n} a$，否则 0。

这个公式在某些情况下会给出武断乃至错误的结果：当 $n < 0$ 时，运算结果为 0，当 a、n 均为 0 时，算出来的 0^0 是 1。

20. $(m \leqslant \forall x \leqslant n) P(x,\overline{y}) = \prod_{x=m}^{n} P(x,\overline{y})$

21. $(m \leqslant \exists x \leqslant n) P(x,\overline{y}) = 1 - \prod_{x=m}^{n} (1 - P(x,\overline{y}))$

$\forall$ 为空真（vacuously true）；$\exists$ 为空假（vacuously false）[⊖]。

22. $(m \leqslant \min x \leqslant n) P(x,\overline{y}) = m + \sum_{i=m}^{n} \prod_{j=m}^{i} (1 - P(j,\overline{y}))$

如果在 m 与 n 之间有 x 能令谓词成立，那么表达式的值就是最小的 x 值，如果由 m、n 所界定的范围是空，那么表达式的值就是 m，如果不为空，而且范围内的所有 x 均使命题不成立，那么表达式的值就是 $n+1$。该操作称为“有界最小化”（bounded minimalization），是生成新公式函数的强大工具。正如下个公式所示，该操作有些反函数的意味。把各个连乘操作的结果累加起来，以此求出“有界最小化”，这一方法是由 GoodStein 给出的 [Good]。

⊖ 这两个概念可参考：http://en.wikipedia.org/wiki/Vacuous_truth 与 http://www.abstractmath.org/MM/MMConditional.htm。——译者注

23. $\lfloor\sqrt{n}\rfloor=(0\leqslant \min k\leqslant |n|)((k+1)^2>n)$

这就是“整数平方根”函数，为了令其成为“全函数”，我们把 $n<0$ 时的函数值定义为0。

24. $d\mid n=(-|n|\leqslant \exists q\leqslant |n|)(n=qd)$

该式表示“d能整除n”这一谓词，据此，0可以为0所整除，而不等于0的数n则不能为0所整除。

25. $n\div d=$ 若 $n\geqslant 0$ 则$(-n\leqslant \min q\leqslant n)(0\leqslant \exists r\leqslant |d|-1)(n=qd+r)$

$=$ 否则$(n\leqslant \min q\leqslant -n)(-|d|+1\leqslant \exists r\leqslant 0)(n=qd+r)$

这就是常用的“向0取整式整数除法”(truncating form of integer division)。当$d=0$时，用该式算出的结果是$|n|+1$，这显得有些武断。

26. $\mathrm{rem}(n,d)=n-(n\div d)d$

这就是常用的求余函数。若 $\mathrm{rem}(n,d)$ 非0，则其正负号与被除数 n 相同。若 $d=0$，则余数为 n。

27. $\mathrm{isprime}(n)=n\geqslant 2\ \&\ \neg(2\leqslant \exists d\leqslant |n|-1)(d\mid n)$

28. $\pi(n)=\sum_{i=1}^{n}\mathrm{isprime}(i)$

（小于等于 n 的素数个数。）

29. $p_n=(1\leqslant \min k\leqslant 2^n)(\pi(k)=n)$

30. $\mathrm{exponent}(p,n)=(0\leqslant \min x\leqslant |n|)\neg(p^{x+1}\mid n)$

当 $n\geqslant 1$ 时，此式的值为 n 中素因子 p 的指数。

31. 当 $n\geqslant 0$ 时：

$$2^n=\prod_{i=1}^{n}2,\qquad 2^{2^n}=\prod_{i=1}^{2^n}2,\qquad 2^{2^{2^n}}=\prod_{i=1}^{2^{2^n}}2,\text{等等。}$$

32. 在$\sqrt{2}$的十进制小数形式中，小数点后第 n 位数是 $\mathrm{rem}(\lfloor\sqrt{2\cdot 10^{2n}}\rfloor,10)$。

由此可见，公式函数的范围相当大。然而其范围毕竟有限，至少根据下述定理是如此：

定理 若 f 为公式函数，则有常数 k 可令

$$f(\overline{x})\leqslant 2^{2^{\cdots^{2^{\max(|x_1|,\cdots,|x_n|)}}}}$$

成立，式子中有 k 个2。

如果能证明符合本节开头规则1至规则5的公式函数都可使上述定理成立，那么也就证明了此定理。例如，根据规则1，$f(\overline{x})=c$ 是个公式函数，那么就能找到某个 h，使得

$$f(\overline{x})\leqslant 2^{2^{\cdots^{2}}}\}h$$

成立，式子中有 h 个2。因此，由于 $\max(|x_1|,\cdots,|x_n|)\geqslant 0$，所以可得

$$f(\overline{x})\leqslant 2^{2^{\cdots^{2^{\max(|x_1|,\cdots,|x_n|)}}}}\}h+2$$

根据规则 2，$f(\overline{x})=x_i$ 是公式函数，所以 $f(\overline{x})\leqslant\max(|x_1|,\cdots,|x_n|)$ 成立，于是上述定理成立，其中的 k 值为 0。

根据规则 3，假设有如下两个公式函数：

$$f(\overline{x})\leqslant \left.2^{2^{\cdots^{2^{\max(|x_1|,\cdots,|x_n|)}}}}\right\}k_1 \text{ 及 } g(\overline{x})\leqslant \left.2^{2^{\cdots^{2^{\max(|x_1|,\cdots,|x_n|)}}}}\right\}k_2。$$

那么，显然可知

$$\begin{aligned} f(\overline{x})\pm g(\overline{x}) &\leqslant 2\cdot \left.2^{2^{\cdots^{2^{\max(|x_1|,\cdots,|x_n|)}}}}\right\}\max(k_1,k_2) \\ &\leqslant \left.2^{2^{\cdots^{2^{\max(|x_1|,\cdots,|x_n|)}}}}\right\}\max(k_1,k_2)+1。\end{aligned}$$

同理可证，定理对 $f(x,y)=xy$ 这样的公式函数也成立。

亦可证明定理对规则 4 与规则 5 成立，其过程稍显乏味，但也不难，此处从略。

根据此定理，函数

$$f(x)=\left.2^{2^{\cdots^{2^x}}}\right\}x \tag{4}$$

不是公式函数，因为无论常数 k 为何值，只要 x 足够大，(4) 式的值总会超过定理中的那个表达式。

如果对递归函数论感兴趣，笔者可以告诉您，等式 (4) 是一种“原始递归函数”(primitive recursive function)[⊖]。而且，由原始递归函数的定义出发，很容易就能证明公式函数也是原始递归函数。因此，公式函数是原始递归函数的“真子集” (proper subset)。欲研究此问题的读者可参阅 [Cut] 一书。

总之，本节想要说明的是，不仅可以用初等函数构造出生成第 n 个素数的公式，而且也能用其构造出一大批数学中常见的其他函数。而且，本节所讲的“公式函数”，其公式里没有使用三角函数、向下取整函数、绝对值函数、底数为 −1 的幂函数，甚至连除法都不曾用到。唯一可以指摘的地方就是，在构造谓词 $a=b$ 的求值公式时，使用了“若连乘操作中任何一个因子为 0，则乘积为 0”这一事实。

诚然，看到了这些公式之后，你会觉得它们实在很没意思。大家还在继续努力寻找“有趣的”素数公式。例如，[Rib] 一书引用了一个令人惊叹的定理，此定理由 W·H·米尔斯 (W. H. Mills) 于 1947 年证明，其内容是，当 n 为大于等于 1 的任意整数时，能找到一个 θ 值，令下列表达式恒为素数：

$$\left\lfloor \theta^{3^n} \right\rfloor$$

实际上，有无数个这样的 θ（例如 1.306 377 883 8+ 与 1.453 750 862 548 3+）。此外，“3”这个数也没有特别之处，用大于等于 2.106 的任意实数来替换 3，定理依然成立（只是值 θ 不同而已）。要是能证明 n^2 与 $(n+1)^2$ 之间总有一个素数就更好了，可以直接把 3 改成 2。这一论述几乎可以成立，然而现在还没人能证明出来。对此类算式感兴趣的读者请参阅 [Rib] 与 [Dud]，里面有许多更为奇妙的公式。

⊖ 也称“递归原语函数”，详情参见：http://zh.wikipedia.org/wiki/原始递归函数。——译者注

18.5 习题

1. 请证明：对任意整数系数的非常数多项式 $f(x)$ 来说，能令 $|f(x)|$ 为合数的 x 值有无数个。
提示：若 $f(x_0)=k$，则考虑 $f(x_0+rk)$ 的取值，其中 r 为大于1的整数。
2. 证明威尔逊定理：对于 $p>1$ 的整数来说，当且仅当
$$(p-1)!\equiv -1 (\bmod p)$$
成立时，p 才是素数。
提示：为了证明当 p 为素数时 $(p-1)!\equiv -1 (\bmod p)$ 成立，可以把阶乘展开，然后将符合 $ab\equiv 1 (\bmod p)$ 的 (a, b) 数值对编为一组。使用10.16节的定理MI。
3. 请证明，若 n 为大于等于4的合数，则下式成立：
$$(n-1)! = 0(\bmod n)$$
4. 估算出满足米尔斯定理的 θ 值，并在其过程中给出该定理的非正式证明。假设我们预先知道：当 $n>1$ 时，n^3 与 $(n+1)^3$ 之间总有一个素数。（这一论述依赖于黎曼猜想[㊀]，然而在n值足够大时，无需借助黎曼猜想就已经可以单独证明此命题了。）
5. 假设有个集合，其中数字都具备 $a+b\sqrt{-5}$ 的形式（a、b 均为整数）。那么请证明：2与3是此集合中的“素数”，也就是说，无法在该集合中使用除 ± 1 之外的数（± 1 是该集合的“单位元素”(unit)）写出2和3的因数分解式。在集合中找到一个具备两种质因子分解式的元素。（根据“算术基本定理”[㊁]，在不考虑“单位元素”[㊂]和因子之幂[㊃]的情况下，每个正整数的质因子分解式只有一种写法。而本题所讲的这个集合，其元素之间的“加法”与“乘法”都按复数规则执行，该集合元素的质因子分解式并不唯一。此集合就是数学中所说的“环”[㊄]。）

㊀ Riemann Hypothesis，简称RH，由德国数学家波恩哈德·黎曼（Bernhard Riemann，1826—1866）于1859年提出，是数学中尚未解决的重要问题。详情参见：https://zh.wikipedia.org/wiki/黎曼猜想。——译者注

㊁ fundamental theorem of arithmetic，又称“正整数唯一分解定理”，即每个大于1的自然数均可写为质数的积，而且这些素因子按大小排列之后，仅有一种写法。详情参见：http://zh.wikipedia.org/wiki/算术基本定理。——译者注

㊂ 在此语境下指自然数1。——译者注

㊃ 比如说，72的质因子分解式为 $2^3\times 3^2$（也就是 $2\times2\times2\times3\times3$），那么其中的 2^3（也就是 $2\times2\times2$）与 3^2（也就是 3×3）必须视为一个整体，不能展开后混排为 $2\times3\times2\times2\times3$、$3\times2\times2\times2\times3$ 等形式。——译者注

㊄ ring，是一种代数结构，其“加法”和“乘法”操作扩展了通常意义下的加法与乘法。自然数是“唯一分解整环”(Unique factorization domain，UFD)，而习题中的 $a+b\sqrt{-5}$ 又称 $\mathbf{Z}[\sqrt{-5}]$ 是一种“非唯一分解整环”，也就是作者说的“质因子分解式并不唯一”。详情参见：http://zh.wikipedia.org/wiki/环_(代数)与http://zh.wikipedia.org/wiki/唯一分解整环。——译者注

参考答案

第1章

1. 可以用以下 while 语句精确表述 for 循环的意思：

```
e1;
while (e2) {
  statement
  e3;}
```

若 for 循环中没有 e_2，那么就将上文中 while 的循环条件写为 1（除非 statement 中有语句能终止循环，否则就一直运行下去）。

用 do 循环来转写 for 循环是件麻烦事，因为 do 的循环体总要执行一次才行，而 for 的循环体可能连一次都不会执行，这要根据 e_1 与 e_2 的值来判断。即便如此，我们还是可以把 for 循环改写为如下形式：

```
e1;
if (e2) {
  do {statement; e3;} while (e2);
}
```

如果 for 中不存在 e_2，那么与 while 循环一样，用 1 代替即可。

2. 要是把代码写成：

```
for (i = 0; i <= 0xFFFFFFFF; i++) {...}
```

那么就成了个死循环[⊖]。下面这个循环是可以正常运行的：

⊖ 因为当 i 增长至 0xFFFFFFFF 时，执行完循环体，i 会自增 1，这样又回到 0，于是下一轮循环开始之前的判别式 i<=0xFFFFFFFF 仍然成立，导致循环无法终止。如果要用 for 循环做，可以写成：

```
for (i=0; i<0xFFFFFFFF; i++) {
 ...
}
 ...
```

——译者注

```
i = 0xFFFFFFFF;
do {i = i + 1; ...} while (i < 0xFFFFFFFF);
```

3. 文中提到的乘法指令就是这样，两个32位元数的乘积是个64位元数，它需要输出到两个寄存器才行。
书里还说了除法操作，在CPU中执行这个指令时，通常余数也会和商一并算出，对于那种同时需要这两个值的程序来说，能节省执行时间。
实际上，现实环境中的很多电脑在执行除法操作时，都是用双字做被除数，用单字做除数，并且同时算出商和余数。这样的话，就需要3个源寄存器和两个目标寄存器了。
索引定址存储指令[㊀]也会用到3个源寄存器，它们分别表示待存储的寄存器、基地值寄存器（base register）、索引寄存器（index register）。
很多CPU都提供了高效处理寄存器位段的extract与insert指令。extract指令的通用形式需要3个源寄存器和1个目标寄存器。3个源寄存器分别表示：含有待提取位段的字组、开始提取位元的位置、提取操作终止的位置或者提取的长度。提取结果将靠右对齐，以0或符号位填充空缺，并存于目标寄存器中。有些CPU只支持用常数来表示提取位段长度，这个折中方案是合理的，因为通常大家也是这样用的。
通用的位段插入指令需要读取4个源寄存器并写入1个目标寄存器。支持此指令的CPU常常用寄存器来表示操作数。这几个操作数的含义分别是：含有待插入位段的源字组（位段从权重最低的一侧算起）、目标寄存器的起始插入位置、插入位段的长度。除了读取上述3个源寄存器外，还需要读取目标寄存器，因为指令要将其内容与待插入的位段组合，并重新写回此寄存器。与位段提取指令一样，位段长度通常也用常量表示，这样一来，只需读取3次寄存器并写入1次即可。
某些CPU还有一系列选择指令（select）：

```
SELcc  RT,RA,RB,RC
```

首先检查RC的值是否满足操作码中指定的条件（也就是指令中的cc部分，它可以是EQ、GT、GE等[㊁]），若满足，则将RA“选中”，若不满足，则“选中”RB。然后，把上一步中“选中”的那个寄存器值复制到目标寄存器中。
很多人都没发现这个不太常用的位选择指令（bit select，也叫multiplex，数据选择指令）其实也需要3个源寄存器：

```
MUX  RT,RA,RB,RC
```

此处RC为掩码。若对应位元是1，则把RA中相应的位元选中，不然就选取RB中相应的位元，所有选好的位元最后都按其位置存入RT。也就是说，该指令与下述操作

㊀ Index Store Instruction，详情参见：http://pic.dhe.ibm.com/infocenter/aix/v7r1/index.jsp?topic=%2Fcom.ibm.aix.aixassem%2Fdoc%2Falangref%2Fidalangref_stdcx_instrs.htm。——译者注

㊁ 分别表示等于（equal to）、大于（greater than）、大于等于（greater than or equal to）。——译者注

等效：

```
RT <-- RA & RC | RB & ~RC
```

有时双精度浮点数（double）的左、右移位指令也很有用：

```
SHLD  RT,RA,RB,RC
```

该指令将 RA 与 RB 连起来作为一个 double 长寄存器，然后将其按 RC 中给定的量向左移位（或向右移位）。RA 与 RB 移位后的结果，只有一部分能容纳于 RT 中。此类指令通常在高精度计算[1]及更偏向实用技术的场合中使用。

在信号处理（signal processing）与其他应用领域中，计算 $A*B+C$ 的指令很有用，这种指令对整数及浮点数均适用。

当然了，多字加载（load multiple）与多字存储（store multiple）指令[2]也需要执行多次寄存器读取及写入操作。尽管很多 RISC 指令集中都有这些指令，但通常不将其视为 RISC 指令。

第 2 章

1. （以下推导过程由 David de Kloet 给出）while 循环的执行次数显然就是 $\boldsymbol{x}$ 末尾“0”的个数。而 k 个“1”会将这个字组中的 n 个位元切成 $k+1$ 段，每一段要么不含“0”，要么含有至少一个“0”。而每一个字组中，“0”位元的个数都是 $n-k$。若含有 n 个“1”的不同字组共有 N（$N=\binom{n}{k}$，不过这个值[3]与本段推理无关）个，那么把每个字组中的上述分段对齐，并将其中 0 的个数汇总，则每个字组对应分段中所含“0”的总个数都一样，均为 $N(n-k)/(k+1)$。于是，每个字组分段平均含有 $(n-k)/(k+1)$ 个“0”。字组最右侧，也就是字组尾部全由“0”所组成的那个分段，其“0”的个数也符合此式。

   举例来说，若 $n=32$ 而 $k=3$，那么循环平均执行次数就是 7.25。在很多电脑中，最少花 3 条指令就能实现 while 循环（逻辑与指令、右移指令、条件转移指令），这样的话，单个循环体最快只要 4 个周期就能执行完。依照这些数据，while 循环的总执行时长平

---

[1] 原文为“bignum” arithmetic，也叫 arbitrary-precision arithmetic、multiple precision arithmetic、infinite-precision arithmetic，是一种程序设计算法，旨在以突破 CPU 字长限制来进行更大范围的计算。详情参见：https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic。——译者注

[2] 这两个指令的详情可参考：http://pic.dhe.ibm.com/infocenter/aix/v7r1/index.jsp?topic=%2Fcom.ibm.aix.aixassem%2Fdoc%2Falangref%2Fidalangref_lmw_lm_instrs.htm 与 http://pic.dhe.ibm.com/infocenter/aix/v7r1/index.jsp?topic=%2Fcom.ibm.aix.aixassem%2Fdoc%2Falangref%2Fidalangref_stmw_sm_instrs.htm。——译者注

[3] N 的值就是 $C(n,k)$，即在 n 个位元中选出 k 个位元，并填上“1”，其值等于 $P(n,k)/C(k,k)$，也就是 $n!/(k!\times(n-k)!)$。其中的惊叹号表示阶乘。——译者注

均是 4×7.25，也就是 29 个周期。这比绝大多数 32 位机执行除法指令所花的时间要少，所以说，de Kloet 算法比 Gosper 算法快，而且随着 k 值增大，de Kloet 算法的优势会更加明显。

2. 与 **1** 取按位与操作之后再左移，也就等于说不管 x 其余位元上的值是多少，只根据其最右方的位元就可以判断出左移操作的移位次数了。因此，只要看看 x 最右侧的位元，就可以确定移位之后的结果到底是 x 还是 $x\ll 1$ 了。由于 x 和 $x\ll 1$ 都可以“从右至左”计算出来，所以“根据 x 最右侧位元在两种算式之间二选一”这样的操作，也是可以“从右至左”求出来的。顺便说一下，$x\ll(x\ \&\ 2)$ 不具备“从右至左的可计算性”，而 $(x\ \&-2)\ll(x\ \&\ 2)$ 却可以。

   另一个例子是函数 x^n，此处规定 x^0 等于 1。该函数不能“从右至左”计算，因为如果 x 是偶数，那么 x^n 的最右侧位元值取决于 x 是不是 **0**，而要想确认这一点，单凭它最右方的那个位元是不行的。但是，假设我们知道一个先决条件（priori）：变量 n 非 0 即 1，那么 x^n 就变成一个可以“从右至左”计算的函数了。同理，$x^{n\&1}$ 也是可以“从右至左”计算出来的，比方说我们可以按照下式判定它的值：

$$x^{n\&1}=(x\ \&-(n\ \&\ 1))+1-(n\ \&\ 1)=\begin{cases}1, & n\text{ 为偶数}\\ x, & n\text{ 为奇数}\end{cases}$$

   请注意，x^n 与左移函数有个相似之处：如果 n 为定值，或 n 是一个值局限于 0、1 两数的变量，那么它就具备“从右至左的可计算性”，反之，如果变量 n 的值不受约束，那么就不能按照“从右至左”的方式计算出来。

3. 根据 2.2 节（g）来计算两数之和：

$$x+y=(x\oplus y)+2(x\ \&\ y)$$

   把等式两端除以 2，即可得 Dietz 公式。此公式不会溢出，因为如果两个整数都能分别容纳在一个字组中，那么其均值也必然能用一个字组表示出来。

   如果我们从 2.2 节（i）式开始推导，那么就可以得出文中所说的另一个公式了，它可以把两个无符号数的均值向上取整。

$$\left\lceil\frac{x+y}{2}\right\rceil=(x\mid y)-((x\oplus y)\overset{u}{\gg}1)$$

4. 用 Dietz 公式分别计算 a、b 及 c、d 的平均值，并对其向下取整，然后对两次计算得到的均值 x 与 y 再求均值，并向下取整，最后补上修正值即可：

$$x=(a\ \&\ b)+((a\oplus b)\overset{u}{\gg}1)$$
$$y=(c\ \&\ d)+((c\oplus d)\overset{u}{\gg}1)$$
$$r=(x\ \&\ y)+((x\oplus y)\overset{u}{\gg}1)$$
$$r=r+((a\oplus b)\ \&\ (c\oplus d)\ \&\ (x\oplus y)\ \&\ 1)$$

   补充修正值的那一步看上去需要 7 个操作，其实只要 4 个。因为其中 3 个按位异或的项已经在前面 3 个式子中算好了。这种修正方案的原理是：如果 a 为奇数，b 为偶数，

或 a 为偶数，b 为奇数，那么 x 的值可能比实际平均值小 1/2，在计算完 x 与 y 的均值之后，这个误差会被减小为 1/4。如果这种“截断误差”[一] 仅发生在计算 x 时，那么第一次算出来的 r 就是正确的。因为在此种情况下，真正的平均值只不过是 $r+1/4$，而我们想求的正是向下取整的平均值（floor average），所以最后的 1/4 直接舍去就好。同理，如果只在计算 y 的时候产生了误差，那么 r 的值也只比真实平均值小了 1/4 而已。此外，第一次计算出来的 r 值有可能会比 x、y 的真实平均值少 1/2。上述 3 种误差效果可能会积累起来。只要积累起来的误差不足 1，那么直接无视就好了，因为计算向下取整的平均值时，本来就不用考虑真实平均值中不足 1 的那个小数部分。但是，若这 3 种误差同时发生，那么积累起来的结果就是 $1/4+1/4+1/2=1$，此时必须再给 r 加 1，才能得出正确的均值。于是就有了最后一行：如果 a、b 两者一奇一偶，且 c、d 两者一奇一偶，同时 x 与 y 也是一奇一偶，那么就把最终结果上调 1——这就是最后那个式子要做的事情。

5. 计算 $x \overset{u}{\leqslant} y$ 所用的那个待简化的表达式为：

$$(\neg x \mid y) \& ((x \oplus y) \mid \neg(y-x))$$

由于表达式中逻辑操作的运算结果只取决于 x 和 y 的第 31 位，所以既然知道 $y_{31}=0$，那么表达式就可以化简为：

$$\neg x \& (x \mid \neg(y-x))$$

然后，再把 $\neg x$ “乘”进括号里面去（根据分配律[二]），可得：

$$\neg x \& \neg(y-x)$$

最后运用德摩根定律，就可以化简为只需 3 条基本指令的表达式了：

$$\neg(x \mid (y-x))$$

（如果把逻辑非指令去掉，那么剩下的两条指令就是比较谓词 $x \overset{u}{>} y$ 的值。）

如果 y 是常数，那么根据恒等式 $\neg u=-1-u$，可以将运用完交换律之后得到的那个式子改写为：

$$\neg x \& (x-(y+1))$$

由于 $y+1$ 的值可以在计算表达式之前先求出来，所以此式也可以用 3 条指令实现。在常数 y 的值较小时，用这种形式来计算比较好，因为可以利用“与立即数相加”指令（add immediate）（此问题由 George Timms 提出）。

6. 如果想让第二次加法也产生进位的话，那么必须保证第一次加法带出来的进位是 1，而且第一次求和结果的低 32 位也必须全是“1”。也就是说，第一次的求和结果最少得是

[一] truncation error，又叫截短误差、舍位误差、舍项误差，指把不足 1 的部分舍去。详情参见：http://en.wikipedia.org/wiki/Truncation_error。——译者注

[二] 即 $a \& (b \mid c)=(a \& b) \mid (a \& c)$，布尔代数的各种知识请参考：http://zh.wikipedia.org/wiki/布尔代数。——译者注

$2^{33}-1$。而两个操作数的最大值都是 $2^{32}-1$，两者之和的最大值是 $2^{33}-2$，所以，第二次加法无论如何也不会再产生最高位的进位了。

7. 为了便于表述，我们假定计算机的字组长度为 4。设 x 与 y 是无符号的 4 位二进制整数，设 $f(x,y)$ 是用 4 位加法器对这两个整数做普通二进制加法并考虑“首尾循环进位”之后所产生的 4 位元结果，那么：

$$f(x,y)=\operatorname{mod}\left(x+y+\left\lfloor\frac{x+y}{16}\right\rfloor,16\right)$$

表中列出了将各种 4 位元字组以反码解读之后的结果。如果把一个字组视为无符号数时它的值是 x，那么下列函数可以算出将其按反码解读之后的值：

$$\operatorname{ones}(x)=\begin{cases}x, & 0\leqslant x\leqslant 7\\ x-15, & 8\leqslant x\leqslant 15\end{cases}$$

x	ones（x）
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	−7
1001	−6
1010	−5
1011	−4
1100	−3
1101	−2
1110	−1
1111	−0

现在，必须要证明：如果把 $f(x,y)$、x、y 这 3 个值都视为以反码形式表示的整数，那么 $f(x,y)$ 就是 x 与 y 之和。也就是说，必须证明下式成立：

$$\operatorname{ones}(x)+\operatorname{ones}(y)=\operatorname{ones}(f(x,y))$$

此处只关心结果不溢出时的情况（也就是说，我们假定两反码的求和结果依然能容纳于一个反码中）。

首先考虑第 1 种情况：$0\leqslant x$，$y\leqslant 7$。此时 $\operatorname{ones}(x)+\operatorname{ones}(y)=x+y$，并且

$$f(x,y)=\operatorname{mod}(x+y+0,16)=x+y$$

如果结果不溢出，则其反码必在 0 至 7 之间。而查表可知，在反码之和不溢出时，$x+y$ 必定小于等于 7，于是，根据 ones 函数的定义，在这种情况下，$\operatorname{ones}(x+y)=x+y$。

第 2 种情况：$0\leqslant x\leqslant 7$，$8\leqslant y\leqslant 15$。由于 $\operatorname{ones}(x)\geqslant 0$ 而 $\operatorname{ones}(y)\leqslant 0$，所以不可能溢出。在此情况下，$\operatorname{ones}(x)+\operatorname{ones}(y)=x+y-15$，如果 $x+y<16$，那么

$$f(x,y)=\operatorname{mod}(x+y+0,16)=x+y$$

在这种情况下，由于 $x+y$ 的值至少是 8，所以根据 ones 函数的定义式，可知 $\operatorname{ones}(x+y)=x+y-15$。如果 $x+y\geqslant 16$，那么

$$f(x,y)=\operatorname{mod}(x+y+1,16)=x+y+1-16=x+y-15$$

因为 $x+y$ 最大值为 22，最小值为 16，所以 $1\leqslant x+y-15\leqslant 7$，于是根据 ones 函数定义式可知：$\operatorname{ones}(x+y-15)=x+y-15$。

第 3 种情况：$8\leqslant x\leqslant 15$，$0\leqslant y\leqslant 7$，此时证明过程与第 2 种情况类似。

第 4 种情况：$8\leqslant x\leqslant 15$，$8\leqslant y\leqslant 15$，此时 $\operatorname{ones}(x)+\operatorname{ones}(y)=x-15+y-15=x+y-30$，而且

$$f(x,y)=\operatorname{mod}(x+y+1,16)=x+y+1-16=x+y-15$$

根据刚才给定的 x、y 取值范围，可以推出 $16\leqslant x+y\leqslant 30$。而此时如果要避免溢出，

那么查表可知，$x+y$ 必须大于等于 23。因为用反码的形式来看，如果－6 加上一个负数之后还不溢出，那么这个负数可以是－1，也可以是－0，但不能是－2。[㊀]因此，可以确定 $23 \leqslant x+y \leqslant 30$，也就是说：$8 \leqslant x+y-15 \leqslant 15$，而根据 ones 函数的定义，可知 ones $(x+y-15)=x+y-30$。

现在谈谈进位能够传播的最大次数问题。执行反码加法时，最极端的进位情况出现在下面这样两个数身上：

```
    111...1111
  + 000...0100
    ----------
    000...0011
  +          1 （首位循环进位，将刚才产生的进位加到求和结果的尾部）
    ----------
    000...0100
```

此时进位传播了 n 次，n 为字组位宽。如果是补码加法，且不考虑最高位产生的进位，那么在最极端的情况，进位会传播 $n-1$ 次。

下面两种现象对比起来很有趣：以 4 个位元的数为例，不论是进行无符号的普通加法还是补码算术，其结果（就算溢出了也无所谓）总等于正确答案与 16 取模之后的那个数；如果以反码形式执行加法，那么其结果不管对不对，总是等于正确答案与 15 取模之后的那个数[㊁]。假设 x_n 是 x 中序号为 n 的位元，那么以二进制补码来表示，$x=-8x_3+4x_2+2x_1+x_0$，如果换用反码表示，那就是 $x=-7x_3+4x_2+2x_1+x_0$。

8. $((\boldsymbol{x} \oplus \boldsymbol{y}) \,\&\, \boldsymbol{m}) \oplus \boldsymbol{y}$

9. $\boldsymbol{x} \oplus \boldsymbol{y}=(\boldsymbol{x} \mid \boldsymbol{y}) \,\&\, \neg(\boldsymbol{x} \,\&\, \boldsymbol{y})$

10. [Arndt] 如果 $\boldsymbol{x}$ 中 $\boldsymbol{i}$、$\boldsymbol{j}$ 两个位置上的位元值不同，那么变量 $\boldsymbol{t}$ 就是 1（整个算法需 6 条指令）。

$$\boldsymbol{t} \leftarrow ((\boldsymbol{x} \overset{u}{\gg} \boldsymbol{i}) \oplus (\boldsymbol{x} \overset{u}{\gg} \boldsymbol{j})) \,\&\, \mathbf{1}$$
$$\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus (\boldsymbol{t} \ll \boldsymbol{j})$$

如果想交换 $\boldsymbol{i}$、$\boldsymbol{j}$ 两个位置上的位元，只需再执行 $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus (\boldsymbol{t} \ll \boldsymbol{j})$ 即可。

11. 正文中说过，任何布尔函数 $f(x_1, x_2, \cdots x_n)$ 都可以分解成 $g(x_1, x_2, \cdots x_{n-1}) \oplus x_n h(x_1, x_2, \cdots x_{n-1})$ 的形式。设具有 n 个自变量的布尔函数最终可由此公式分解为 $c(n)$ 个二元布尔操作，且 $n \geqslant 2$，那么

$$c_{n+1}=2c_n+2$$

㊀ 根据 ones 函数定义反推，－6 对应的无符号值是 9，而－2 对应的无符号值是 13，二者之和为 22，可见小于等于 22 都会导致溢出，所以原作者才说："$x+y$ 必须大于等于 23"。——译者注

㊁ 例如两个无符号数分别是 8(1000) 和 15(1111)，那么两数之和的正确结果为 23(10111)，这个数无法用 4 个位元容纳下，于是最高位会被丢弃，从而成了 7(0111)，这个结果虽然不对，但是它却总是等于正确答案（即 23）和 16 的模。同理，假设两个反码数为－7(1000) 和－3(1100)，那么两数之和的正确结果为－10(10101)，但它无法用 4 个位元容纳，所以会被截短为 5(0101)，这个值虽然不对，但是它却等于正确答案（即－10）和 15 取模后的结果。——译者注

其中已知 $c_2=1$。那么根据此递推关系式，可以求出[一]

$$c_n = 3\times 2^{n-2}-2$$

（这只不过是理论上的最大值罢了，实际上实现布尔函数分解所花的最大指令数要远小于此。）

12. (a) 式证明过程如下：

$$\begin{aligned} f(x,y,z) &= \bar{z}f_0(x,y)+zf_1(x,y)\\ &= \bar{z}f_0(x,y)\oplus zf_1(x,y)\\ &= \bar{z}f_0(x,y)\oplus(1\oplus\bar{z})f_1(x,y)\\ &= \bar{z}f_0(x,y)\oplus f_1(x,y)\oplus\bar{z}f_1(x,y)\\ &= f_1(x,y)\oplus\bar{z}(f_0(x,y)\oplus f_1(x,y)) \end{aligned}$$

由上述形式可知 (a) 式成立。

(b) 式证明过程如下：

根据 (a) 式可得

$$\begin{aligned} f(x,y,z) &= f_1(x,y)\oplus\bar{z}(f_0(x,y)\oplus f_1(x,y))\\ &= \overline{\overline{f(x,y)}}\oplus\overline{\overline{\bar{z}(f_0(x,y)\oplus f_1(x,y))}}\\ &= \overline{\overline{f(x,y)}}\oplus(z+\overline{(f_0(x,y)\oplus f_1(x,y))}) \end{aligned}$$

由上述形式可知 (b) 式成立。

13. 根据 2.23 节中表 2.3 的书写方式，剩下几个布尔函数分别是：0000＝andc(x，x)、0011＝and(x，x)、0100＝andc(y，x)、0101＝and(y，y)、1010＝nand(y，y)、1011＝cor(y，x)、1100＝nand(x，x)、1111＝cor(x，x)。

14. 剩下的 10 个函数没办法用少于 8 种类型的布尔指令实现出来。因为那 10 个名副其实的布尔函数以数字形式写出，分别是：

0001	0010	0100	0110	0111
1000	1001	1011	1101	1110

要实现 0010 这个函数，可以把 0100 这个函数[二]的两个自变量互换位置，于是就省下了一条指令，同理，把 1011 这个函数的两个自变量换位，就可以实现出 1101 函数[三]了。在这 10 个函数中，只有上面两对函数才符合交换律，所以只有这 4 个函数可以通过交换操作数的方式归并为两条指令，而剩下 6 条指令则没有办法再归并了。因为如果采用另外一种归并布尔函数的办法，把指令的两个操作数填成一个相同的值，那

[一] 可以把 $c_{n+1}=2c_n+2$ 变形为 $c_{n+1}+2=2c_n+4=2(c_n+2)$，由此可以看出，$c_n+2$ 是一个以 c_2+2 为首项，以 2 为公比的等比数列，故 $c_n+2=(c_2+2)\times 2^{n-2}=3\times 2^{n-2}$，所以 $c_n=3\times 2^{n-2}-2$。——译者注

[二] 0010 表示 $\neg x\ \&y$，而 0100 表示 $x\ \&\neg y$，由于&操作符的两个操作数互换位置后结果不变，故可以通过互换变量位置的方式，用其中一个来实现另一个。——译者注

[三] 1011 表示 $\neg x\mid y$，而 1101 表示 $x\mid\neg y$，由于 | 操作符的两个操作数互换位置后结果不变，故可以通过互换变量位置的方式，用其中一个来实现另一个。——译者注

么只会把一个真正的二元布尔函数退化为一元布尔函数或常数。所以，无论如何都至少需要 8 种类型的布尔指令才能实现这 10 个函数。

15. 下标列出了实现 16 种布尔函数所需的 6 条指令。其中 x 表示寄存器操作数的内容，而 k 表示操作码中常量位段的值。
表中未列的 10 种布尔函数，实现方式如下：0000=and(x,0)、0010=and($x,\bar{k}$)、0011=or(x,0)、0100=nor($x,\bar{k}$)、1001=xor($x,\bar{k}$)、1010=const($x,\bar{k}$)、1011=or($x,\bar{k}$)、1100=nor(x,0)、1101=nand($x,\bar{k}$)、1111=nand(x,0)。

实现 16 种布尔函数所必备的 6 条 R-I 指令

函数值	公式	指令助记符
0001	xk	and
0111	$x+k$	or
0110	$x\oplus k$	xor
1110	$\overline{xk}$	nand
1000	$\overline{x+k}$	nor
0101	k	const

16. 笔者不知道如何以正常推理的形式证明此问题。不过只要编一段程序，就可以把只用 3 条指令就能实现的全部三变量布尔函数找出来。下面就用 C 语言来写这段程序。为了给出令人信服的答案，我们把程序写得尽量简单易懂一些。当然还是有很多地方可以优化的，这个问题后面也要谈到。
程序采用长度为 8 的位串来构建函数的真值表（truth table），其中 x、y、z 的取值顺序也按照通常真值表中的顺序来定[⊖]。程序中有一个长度为 256 字节的数组，程序刚开始执行时，它的每个元素都是 0；运行程序之后，每生成一个三元布尔函数，就把数组 found 中相应的元素置 1。
程序所用的真值表如下所示。

三变量布尔函数真值表

$f_0=x$	$f_1=y$	$f_2=z$	f_3	f_4	f_5
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

⊖ 意思也就是，每一行所对应的 x、y、z 取值分别为：000、001、010、011、100、101、110、111。——译者注

fun 数组中的每一个元素都表示真值表中这 6 栏所对应的函数。fun 的前三个元素就是真值表中前 3 栏的函数码，这些函数码用十六进制写出来就是 0F、33、55，分别表示 3 个非常简单的三变量布尔函数：$f(x,y,z)=x$、$f(x,y,z)=y$、$f(x,y,z)=z$。接下来的 3 个元素表示真值表中后 3 栏，也就是当前正在试验的 3 个函数，它们分别代表用 1 条、2 条、3 条二元布尔指令所能拼合出来的 3 变量布尔函数。

从概念上讲，程序有三重循环，每一重都表示当前正在尝试的一种二元布尔指令。最外层循环在 x、y、z 3 个函数中任取两个，然后遍历全部 16 种二元布尔操作（总循环次数是 16 * 3 * 3=144）。每执行一次循环体，它就把当前要试验的这一种操作指令套用在从 x、y、z 中选出来的那两个 8 位元操作码上，并把运算结果放在 fun [3] 中。

接下来一重循环与刚才类似，也是要尝试全部 16 种二元布尔操作，不过它要从 x、y、z 以及最外层循环刚算出来的 fun [3] 这 4 者中选取两个作为操作数（这样的话，每次从最外围循环进来都要迭代 16 * 4 * 4=256 次）。每次循环体尝试出来的结果放在 fun [4] 中。

最内层循环与刚才两重类似，还是得尝试全部 16 种二元布尔操作，然而两个操作数要从 x、y、z 以及刚才两重循环算好的 fun [3]、fun [4] 这 5 个函数中选择（所以每次进入最内层循环，都要迭代 16 * 5 * 5=400 次）。一旦算好这样一个函数 fun [5]，那么就把 found 数组中下标与该函数操作码相同的那个元素置 1。

最后，程序把 found 数组中的元素输出成 16 行，每行 16 个元素。found 数组很多位置上的元素值都是 0，这说明只靠 3 个二元布尔指令是不足以实现全部 256 种三变量布尔函数的。第一个实现不了的函数就是操作码为 0x16（二进制 0010 0110）的那个函数，它的定义式是：$f(x,y,z)=\bar{x}yz+x\bar{y}z+xy\bar{z}$。

利用对称性，可以简化循环次数。举例来说，如果给定一个操作符 op 及两个操作数 x、y，那么不需要分别计算 op（x，y）与 op（y，x）的值，因为只要 op（x，y）算出来了，那么 16 种布尔函数里面肯定还能找到一个操作符 op′，使得 op′（x，y）的值就是 op（y，x）的值。同理，op（x，x）的值其实也不用算，因为 16 种布尔函数里面肯定能找到一个操作符 op′，使得 op′（x，y）的值就是 op（x，x）。因此，尝试各种操作数的那个最外层循环可以改写为：

```
for (i1 = 0; i1 < 3; i1++) {
for (i2 = i1 + 1; i2 < 3; i2++) {
```

其他两重循环也可以据此改写。

还有一个地方可以改进，那就是不必把表 2.3 中的 16 种二元布尔函数全都尝试一遍。其中第 0、3、5、10、12、15 栏的这 6 种操作可以省略，这样就能把尝试的次数由 16 次降低为 10 次。此一论述的证明过程稍显冗长，此处从略。

这个程序可以用来做多种试验，如可以把指令集缩小，把所允许的最大指令个数⊖ 放宽，增加布尔函数中的变量个数，等等。不过试验前要注意：由于主程序中的循环次数是由指令个数决定的，所以程序执行时间会随着指令数急剧增大。从现实角度讲，允许的最大指令数不应超过 5。

```
/* Determines which of the 256 Boolean functions of
three variables can be implemented with three binary
Boolean instructions if the instruction set includes
all 16 binary Boolean operations. */

#include <stdio.h>

char found[256];

unsigned char boole(int op, unsigned char x,
                            unsigned char y) {
   switch (op) {
      case  0: return 0;
      case  1: return x & y;
      case  2: return x & ~y;
      case  3: return x;
      case  4: return ~x & y;
      case  5: return y;
      case  6: return x ^ y;
      case  7: return x | y;
      case  8: return ~(x | y);
      case  9: return ~(x ^ y);
      case 10: return ~y;
      case 11: return x | ~y;
      case 12: return ~x;
      case 13: return ~x | y;
      case 14: return ~(x & y);
      case 15: return 0xFF;
   }
}
#define NB 16           // Number of Boolean operations.
int main() {

   int i, j, o1, i1, i2, o2, j1, j2, o3, k1, k2;
   unsigned char fun[6];// Truth table, 3 columns for
                        // x, y, and z, and 3 columns
                        // for computed functions.

   fun[0] = 0x0F;       // Truth table column for x,
   fun[1] = 0x33;       // y,
   fun[2] = 0x55;       // and z.

   for (o1 = 0; o1 < NB; o1++) {
   for (i1 = 0; i1 < 3; i1++) {
   for (i2 = 0; i2 < 3; i2++) {
      fun[3] = boole(o1, fun[i1], fun[i2]);
      for (o2 = 0; o2 < NB; o2++) {
      for (j1 = 0; j1 < 4; j1++) {
      for (j2 = 0; j2 < 4; j2++) {
         fun[4] = boole(o2, fun[j1], fun[j2]);
         for (o3 = 0; o3 < NB; o3++) {
         for (k1 = 0; k1 < 5; k1++) {
```

找出所有能够以 3 条布尔指令实现的三元布尔函数

⊖ 如我们想找出所有能够用 3 条基本布尔指令实现的三变量布尔函数，那么所允许的最大指令数就是 3，如果要找出能用 4 条基本布尔指令实现的三变量布尔函数；那么所允许的最大指令数就是 4。

（续）

```
         for (k2 = 0; k2 < 5; k2++) {
            fun[5] = boole(o3, fun[k1], fun[k2]);
            found[fun[5]] = 1;
         }}}
      }}}
   }}}
   printf("   0 1 2 3 4 5 6 7 8 9 A B C D E F\n");
   for (i = 0; i < 16; i++) {
      printf("%X", i);
      for (j = 0; j < 16; j++)
         printf("%2d", found[16*i + j]);
      printf("\n");
   }
   return 0;
}
```

第 3 章

1. (a) $(\boldsymbol{x}+\mathbf{4})\,\&\,-\mathbf{8}$。

   (b) $(\boldsymbol{x}+\mathbf{3})\,\&\,-\mathbf{8}$。

   (c) $(\boldsymbol{x}+\mathbf{3}+((\boldsymbol{x}\overset{u}{\gg}\mathbf{3})\,\&\,\mathbf{1}))\,\&\,-\mathbf{8}$。

   如果计算机有“位提取”指令（extract instruction），那么（c）式仅需 4 条指令即可。因为只用 1 条“位提取”指令即可算出 $(\boldsymbol{x}\overset{u}{\gg}\mathbf{3})\,\&\,\mathbf{1}$ 的值。

   提示：在计算一大批随机整数的平均值时，若先采用“无误差舍入法”对计算数值舍入，再求舍入后的平均值，则与其他两种舍入方式相比，此方式可以最大限度保持计算结果的准确性。

2. (a) 部分的标准解法是 $((\boldsymbol{x}+\mathbf{5})\overset{u}{\div}\mathbf{10})*\mathbf{10}$。如果计算机支持求余功能，那么也可以用 $\boldsymbol{x}+\mathbf{5}-\mathrm{remu}(\boldsymbol{x}+\mathbf{5},\mathbf{10})$ 计算。这样的话，可以少算一次乘法，但是得多用一条加法指令。

   (b) 部分的解法与（a）类似，只是将其中的 5 换成 4 即可。

   如果一个数是 10 的奇数倍，那么它必然是 2 的奇数倍。基于这一事实，可以推出（c）部分的解法。

```
r = x % 10;
y = x - r;
if (r > 5 | (r == 5 & (y & 2) != 0)
   y = y + 10;
```

   还有一种解法是（其中 $\boldsymbol{x}\leqslant 2^{32}-6$）：

```
r = (x + 5)%10;
y = x + 5 - r;
if (r == 0 & (y & 2) != 0)
   y = y - 10;
```

3. 下面列出一种 C 语言实现代码供参考。

```
int loadUnaligned(int *a) {
   int *alo, *ahi;
   int xlo, xhi, shift;

   alo = (int *)((int)a & -4);
   ahi = (int *)(((int)a + 3) & -4);
   xlo = *alo;
   xhi = *ahi;
   shift = ((int)a & 3) << 3;
   return ((unsigned)xlo >> shift) | (xhi << (32-shift));
}
```

第 4 章

1. 将 $\boldsymbol{a}=\boldsymbol{c}=\mathbf{0}$ 代入不等式（5），可得

$$\text{若} -d<0 \text{ 且 } b\geqslant 0,\quad \mathbf{0}\overset{u}{\leqslant}\boldsymbol{x}-\boldsymbol{y}\overset{u}{\leqslant}\mathbf{2^{32}-1}$$

$$\text{否则}，\quad -\boldsymbol{d}\overset{u}{\leqslant}\boldsymbol{x}-\boldsymbol{y}\overset{u}{\leqslant}\boldsymbol{b}$$

因为其中各数均无符号，所以 $-d<0$ 也就等同于 $d\neq 0$，而 $b\geqslant 0$ 这一条件总是成立，于是，上述不等式可化简为：

$$\text{若 } d\neq 0,\ \mathbf{0}\overset{u}{\leqslant}\boldsymbol{x}-\boldsymbol{y}\overset{u}{\leqslant}\mathbf{2^{32}-1}$$

$$\text{若 } d=0,\ \mathbf{0}\overset{u}{\leqslant}\boldsymbol{x}-\boldsymbol{y}\overset{u}{\leqslant}\boldsymbol{b}$$

如果 $\boldsymbol{d}=\mathbf{0}$，那么很容易就能看出来 $\boldsymbol{y}=\mathbf{0}$，此时显然可得 $0\leqslant\boldsymbol{x}-\boldsymbol{y}\leqslant\boldsymbol{b}$。另一方面，如果 $\boldsymbol{d}\neq\mathbf{0}$，那么当 $\boldsymbol{x}=\boldsymbol{y}=\mathbf{0}$ 时，二者之差是 $\mathbf{0}$；而当 $\boldsymbol{x}=\mathbf{0}$ 且 $\boldsymbol{y}=\mathbf{1}$ 时，二者之差是最大的无符号数。

2. 若 a = 0，则 if (temp> = a) 这条检测语句总是成立，因此，在找到了（从左往右数）第 1 个 b 与 d 的位元值均为“1”的位置后，程序就可以直接把 b 的这一位设成 0，并把其右侧位元设为 1。然后，将修改好的 b 值与 d 做“或”运算，并返回计算结果。我们可以将上述算法改写为下列代码，并用其替换掉整个函数体中原有的代码，这样就可以进一步简化此函数。只有当计算机用的是诸如 Intel x86 系列的“模 32”移位的 CPU 时，才会用到下列代码中的 if 语句。

```
temp = nlz(c & d);
if (temp == 0) return 0xFFFFFFFF;
m = 1 << (32 - temp);
return b | d | (m - 1);
```

比方说，假设：

$$\mathbf{0}\leqslant\boldsymbol{x}\leqslant\mathbf{0b0100\ 1000}，\text{且}$$

$$\mathbf{0b0000\ 0011}\leqslant\boldsymbol{y}\leqslant\mathbf{0b0010\ 1010}$$

那么在寻找 $\boldsymbol{x}\mid\boldsymbol{y}$ 最大值的过程中，上述算法会从左至右扫描 $\boldsymbol{b}$、$\boldsymbol{d}$ 的每个位元，直到发现两者均为“1”的情况。此时，我们可以对 $\boldsymbol{b}$、$\boldsymbol{d}$ 两数在该位置左侧的那些位元做

“或”运算，然后把该位置及其右侧位元设为“1”，再把这两段拼合起来[⊖]，于是就得出了 $b \mid d$ 的最大值。比如在本例中，这个最大值就是：**0b0100 1000 | 0b0010 1010 | 0b0000 1111＝0b0110 1111**。

第5章

1. Norbert Juffa 先生给出的版本是：

```
int ntz (unsigned int n) {
    static unsigned char tab[32] =
    {   0,  1,  2, 24,  3, 19,  6, 25,
       22,  4, 20, 10, 16,  7, 12, 26,
       31, 23, 18,  5, 21,  9, 15, 11,
       30, 17,  8, 14, 29, 13, 28, 27
    };
    unsigned int k;
    n = n & (-n);         /* isolate lsb */
#if defined(SLOW_MUL)
    k = (n << 11) - n;
    k = (k <<  2) + k;
    k = (k <<  8) + n;
    k = (k <<  5) - k;
#else
    k = n * 0x4d7651f;
#endif
    return n ? tab[k>>27] : 32;
}
```

2. $\boldsymbol{x} \overset{u}{\div} (\boldsymbol{x} \,\&\, -\boldsymbol{x})$。2.1.3节中的“snoob函数”曾经用到过类似算式。
3. 将针对 $\boldsymbol{x}$ 的平行前缀操作记为PP－XOR（$\boldsymbol{x}$），那么，如果原函数是 $\boldsymbol{y}$＝PP－XOR（$\boldsymbol{x}$），则反函数为 $\boldsymbol{x}=\boldsymbol{y} \oplus (\boldsymbol{y} \overset{u}{\gg} \mathbf{1})$。为了说明这一点，假设 $\boldsymbol{x}$ 是4位元数abcd（每个字母均表示1个二进制位），那么：

$$\boldsymbol{y}=\text{PP-XOR}(\boldsymbol{x}) = (a)(a \oplus b)(a \oplus b \oplus c)(a \oplus b \oplus c \oplus d),$$

$$\boldsymbol{y} \overset{u}{\gg} 1 = (1)(a)(a \oplus b)(a \oplus b \oplus c),\text{因此}$$

$$\boldsymbol{y} \oplus (\boldsymbol{y} \overset{u}{\gg} 1) = (a)(b)(c)(d)$$

对于并行后缀算法来说，如果原函数 $\boldsymbol{y}$＝PS-XOR（$\boldsymbol{x}$），那么大家也许能猜到，反函数就是：$\boldsymbol{x}=\boldsymbol{y} \oplus (\boldsymbol{y} \ll 1)$。

第6章

1. 可用如下程序求出最长“全1位串”的长度与位置（参考了Norbert Juffa先生的算法）：

⊖ 从左往右看，b、d 两数在该处的位元值均为“1”的首个位置是第5位，所以将两数前4位上的数值“0100”与“0010”求“或”，得“0110”，然后再将该位置及其右侧位置的位元均写为“1”，也就是“1111”，最后把刚才算好的“0110”与“1111”首尾拼合起来，就可以得出最后答案0b0110 1111。——译者注

```
int fmaxstr1(unsigned x, int *apos) {
   int k;
   unsigned oldx;

   oldx = 0;
   for (k = 0; x != 0; k++) {
      oldx = x;
      x &= 2*x;
   }
   *apos = nlz(oldx);
   return k;
}
```

2. 本章正文最后一段说过，要找这样的位串，可以先把 x 里的“0”向左传播 $n-1$ 位，然后在修改过的新 x 中查找最短“全 1 位串”。6.2 节图 6.5 中的代码是个向左传播位元的好办法，其执行时间是对数级的。（不过本题算法的第二部分是线性的，也就是在修改过的 x 里查找最短“全 1 位串”长度这部分。）算法代码如下，其中假定 $1\leqslant n\leqslant 32$。如果找不到这样的位串，就在 apos 中返回 32。这种情况下函数返回的长度 $n-1$ 应视为“未定义”(undefined)。

```
int bestfit(unsigned x, int n, int *apos) {
   int m, s;

   m = n;
   while (m > 1) {
      s = m >> 1;
      x = x & (x << s);
      m = m - s;
   }
   return fminstr1(x, apos) + n - 1;
}
```

3. 下列代码使用 2.1 节中的公式将最右方的“全 1 位串”置 0。

```
int fminstr1(unsigned x, int *apos) {
   int k, kmin, y0, y;
   unsigned int x0, xmin;

   kmin = 32;
   y0 = pop(x);
   x0 = x;
   do {
      x = ((x & -x) + x) & x;  // Turn off rightmost
      y = pop(x);              // string.
      k = y0 - y;              // k = length of string
      if (k <= kmin) {         // turned off.
         kmin = k;             // Save shortest length
         xmin = x;             // found, and the string.
      }
      y0 = y;
   } while (x != 0);
   *apos = nlz(x0 ^ xmin);
   return kmin;
}
```

设 n 是 x 中“全 1 位串”的个数，则当 $n\geqslant 1$（也就是 $x\neq 0$）时，该函数要执行 $5+11n$ 条指令。前提是代码中“if 测试”成立与不成立机会各半，而 pop($\boldsymbol{x}$) 与 nlz($\boldsymbol{x}$) 都各自算作 1 条指令。修改“if (k<=kmin)”这条测试语句中的不等号以及 kmin 变

量的初始值，即可找出最长“全1位串”，而且也可以在发现多个等长位串时选择返回最左侧的还是最右侧的。对此算法稍加修改，也很容易实现出“最佳适配”函数（“best fit” function）。

4. x 的首个二进制位如果为“1”，则表示此处是某个“全1位串”的起始点，这种概率是0.5，其余各二进制位引领“全1位串”的概率都是0.25（这种二进制位自身必须是“1”，且其左侧二进制位必须是“0”）。因此，“全1位串”的平均个数为 $0.5+31\cdot0.25=8.25$。

5. 可以想见，随着位数增多，在绝大部分二进制数中都能找到长度为1的“全1位串”。只要它以“10”开头，或以“01”结尾，或包含“010”这样的位串，那么最短“全1位串”的长度就是1。因此，最短“全1位串”的平均长度应该略微高于1。

   穷举 2^{32} 种字组取值之后发现，最短“全1位串”平均长度约为1.011 795。

6. （John Gunnels先生的解法）这个问题看着容易做起来难，然而其求解技巧值得了解。此解法基于递归算法，它根据下表将待统计的数值分成4个集合，计算归入每类中的数值个数。表中“单例”（singleton）一词是指长度为1的“全1位串”[⊖]，“nnn”表示长度大于等于0但不含“单例”的位串，“sss”表示长度至少为1且含“单例”的位串。省略号表示其前方位元可出现0次或多次。每个二进制数只能归入下述4个集合之一。

集合	其中数值的二进制形式	描述
A	nnn0… 或 无值（null）	本身不含“单例”，但接下来在其右侧追加一个二进制位后，则有可能出现“单例”
B	nnn01 或 1	本身含“单例”，但接下来在其右侧追加一个二进制位，则有可能令“单例”消失
C	nnn011… 或 11…	本身不含“单例”，接下来在其右侧追加一个二进制位后，依然不含“单例”
D	sss0 或 sss01…	本身含“单例”，接下来在其右侧追加一个二进制位后，依然含有“单例”

算法每步都会在当前二进制数右侧追加一个二进制位。执行完该操作后，新生成的数字可能会归属于另外一个集合，其递推关系式如下。式子中有两种可能：若尾部追加“0”，则归入左侧集合，若追加“1”，则归入右侧集合。

$$A \Rightarrow A \text{ 或 } B$$
$$B \Rightarrow D \text{ 或 } C$$
$$C \Rightarrow A \text{ 或 } C$$
$$D \Rightarrow D \text{ 或 } D$$

例如，二进制数“1101”属于集合 B。若尾部加“0”变为“11010”，则属集合 D。则若加“1”变成“11011”，则属集合 C。

⊖ 也就是孤立出现的“1”，其左右两侧如有位元，必须是“0”。——译者注

设 a_n、b_n、c_n、d_n 分别为集合 A、B、C、D 的元素个数，则 n 步之后（n 是二进制数全长）：

$$
\begin{aligned}
a_{n+1} &= a_n + c_n \\
b_{n+1} &= a_n \\
c_{n+1} &= b_n + c_n \\
d_{n+1} &= b_n + 2d_n
\end{aligned}
$$

之所以有上述递推关系，是因为第 $n+1$ 步集合 A 里面的数一方面由第 n 步中集合 A 里面的数加“0”而来，另一方面由第 n 步中集合 C 里面的数加“0”而来，故 $a_{n+1}=a_n+c_n$。第 $n+1$ 步集合 B 里面的数则只可能由第 n 步中集合 A 里面的数加“1”而来，故 $b_{n+1}=a_n$。其余二式同理。

初始条件为 $a_0=1$、$b_0=c_0=d_0=0$。

用电脑程序很容易就能算出这些递推关系式[㊀]，甚至手算也行。$n=32$ 时的结果为：

$$
\begin{aligned}
a_{32} &= 26\ 931\ 732 \\
b_{32} &= 15\ 346\ 786 \\
c_{32} &= 20\ 330\ 163 \\
d_{32} &= 4\ 232\ 358\ 615 \\
b_{32}+d_{32} &= 4\ 247\ 705\ 401
\end{aligned}
$$

我们关心最后一行，也就是最短“全 1 位串”长度是 1 的 32 位二进制数有多少个。其个数约占全部 32 位二进制数（共有 2^{32} 个）的 98.9%。

那“闭合式解法”又如何做呢？这同样很难。下面我们只粗略给出算法梗概。

设 $e_n=b_n+d_n$，我们要算的正是此值。然后，根据上述“差分方程”，再结合 $a_n+b_n+c_n+d_n=2^n$ 这一事实，可得

$$
\begin{aligned}
e_n &= b_n + d_n \\
&= 2^n - a_n - c_n \\
&= 2^n - a_{n+1}
\end{aligned}
$$

因此，只要找到 a_n 的“闭合式解法”，也就能找到 e_n 的相应解法了。

经过下述步骤，可以将 a_n 写成单变量递推关系式。从前述递推关系式可得：

$$
\begin{aligned}
a_n &= a_{n-1} + c_{n-1} \\
&= a_{n-1} + b_{n-2} + c_{n-2} \\
&= a_{n-1} + a_{n-3} + c_{n-2} \\
&= a_{n-1} + a_{n-3} + a_{n-1} - a_{n-2} \\
&= 2a_{n-1} - a_{n-2} + a_{n-3}
\end{aligned}
$$

㊀ 原文为“difference equation”，即“差分方程”，实际上是递归关系式的一种学术叫法，译文从俗。详情参见：https://en.wikipedia.org/wiki/Difference_equation 及 https://en.wikipedia.org/wiki/Finite_difference。——译者注

现在得出的这个递推关系式可通过"常见运算"解出。过程有些繁冗，此处略去[㊀]。其间需解三次多项式（cubic polynomial）[㊁]方程，求出一个实根及两个共轭虚根。将解出的 a_n 代入 e_n 关系式，即可得出下列约数：

$$e_n \approx 2^n - 0.41150 \cdot 1.7549^{n+1} - (0.29425 - 0.13811i)(0.12256 + 0.74486i)^{n+1} - (0.29425 + 0.13811i)(0.12256 - 0.74486i)^{n+1}$$

不难证明，若 n 为整数，则虚部可消去。（提示：若 x 与 y 互为共轭复数，则 x^n 与 y^n 亦共轭。）

可以写出一个只含实数的公式。上述公式第二项的实部显然小于

$$|0.29425 - 0.13811i| \cdot |0.12256 + 0.74486i|^{n+1}$$

当 $n=0$ 时，上述值为

$$0.32505 \cdot 0.75488 \approx 0.24537$$

而当 $n>0$ 时，其值会更小。e_n 关系式的最后一项也是这样。因此，最后两项之和的实部小于 0.5。根据先前所述，e_n 必为整数，所以可将 e_n 表达式第一项舍入至最近整数，或者说：

$$e_n \approx \lfloor 2^n - 0.41150 \cdot 1.7549^{n+1} + 0.5 \rfloor$$

7. 简言之，把二进制数分成如下 10 个集合可解此题。表中"nnn"所指位串其长度大于等于 0，且不含长度为 1 的最短"全 1 位串"，"ddd"所表示的位串其长度大于等于 2，且包含长度是 2 的最短"全 1 位串"，"sss"所指位串长度大于等于 1，且包含长度为 1 的最短"全 1 位串"。（将"sss"纳入集合分类标准，是为了记录下除最右侧位元外其余位置含有"单例"的二进制数。这种二进制数，其最短"全 1 位串"的长度绝不可能是 2。）省略号表示其左侧位元可重复出现 0 次或多次。

集合	其中数值的二进制形式	集合	其中数值的二进制形式
A	nnn0... 或无值（null）	F	ddd01
B	nnn01 或 1	G	ddd011
C	nnn011 或 11	H	ddd0111...
D	nnn0111... 或 111...	I	sss0
E	ddd0	J	sss01...

㊀ $a_n = 2a_{n-1} - a_{n-2} + a_{n-3}$ 这种 $a_n = c_1a_{n-1} + c_2a_{n-2} + \cdots + c_da_{n-d} (c_d \neq 0)$ 形式的式子，学名叫"常系数线性齐次递推关系式"（linear homogeneous recurrence relation with constant coefficient）。构造"特征方程" $r^d - c_1r^{d-1} - c_2r^{d-2} - \cdots - c_d = 0$ 并求其根 r_1 至 r_d，若无重根，则可得通项公式 $a_n = k_1(r_1)^n + k_2(r_2)^n + \cdots + k_d(r_d)^n$，将递推关系式的 d 个初始值 a_0 至 a_{d-1} 联立为 d 元一次方程组，即可求出其中 k_1 至 k_d 等系数。详情参见：https://en.wikipedia.org/wiki/Difference_equation#Linear_homogeneous_recurrence_relations_with_constant_coefficients 及 http://delta.cs.cinvestav.mx/~debrup/ho1.pdf。——译者注

㊁ 一元三次方程解法参见：https://zh.wikipedia.org/wiki/三次方程。——译者注

每次向当前二进制数右侧追加一个位元后，它就有可能归入另外一个集合，其分类变化情况如下所述。每个递推关系式都有两种情况：若新增的位元是“0”，则归入左侧新集合，若为“1”，则归入右侧新集合。

$$
\begin{array}{ll}
A \Rightarrow A \text{ 或 } B & F \Rightarrow I \text{ 或 } G \\
B \Rightarrow I \text{ 或 } C & G \Rightarrow E \text{ 或 } H \\
C \Rightarrow E \text{ 或 } D & H \Rightarrow E \text{ 或 } H \\
D \Rightarrow A \text{ 或 } D & I \Rightarrow I \text{ 或 } J \\
E \Rightarrow E \text{ 或 } F & J \Rightarrow I \text{ 或 } J
\end{array}
$$

设 a_n，b_n，…，j_n分别为 A、B，…，J 等集合在第 n 步的元素个数（n 是待统计二进制数的总长）。则

$$
\begin{array}{ll}
a_{n+1} = a_n + d_n & f_{n+1} = e_n \\
b_{n+1} = a_n & g_{n+1} = f_n \\
c_{n+1} = b_n & h_{n+1} = g_n + h_n \\
d_{n+1} = c_n + d_n & i_{n+1} = b_n + f_n + i_n + j_n \\
e_{n+1} = c_n + e_n + g_n + h_n & j_{n+1} = i_n + j_n
\end{array}
$$

初始条件是：$a_0=1$，其余变量皆为 0。

我们要算的是最短“全 1 位串”长度为 2 的二进制数有多少个，也就是算 $c_n+e_n+g_n+h_n$ 的值。当 $n=32$ 时，根据递推关系式可求得该值为 44 410 452，它占全部 32 位二进制数的 1.034%。顺便算一下：最短“全 1 位串”长度为 1 的二进制数共有 $b_n+f_n+i_n+j_n$ 个，当 $n=32$ 时，就是 4 247 705 401 个，这印证了上一道题的答案。

本书对此问题就研究到这里。

第 7 章

1. 在未反转普通整数中，只要对一定数量的连续低权重位元取反，即可求出该数递增后的值[⊖]。例如要给 **0x321F** 加 1，只需将原数和掩码 **0x003F** 求“异或”即可。同理，要“递增”一个位元反转之后的数，只需对其中某些高权重位元取反即可，取反使用的掩码以“全 1 位串”起头，后面跟着“全 0 位串”。Möbius 公式就是计算出这个掩码并将它应用于位元反转之后的整数上。（本书正文里使用 nlz 操作的那个算法也是如此。）

   递增未反转的整数时所使用的掩码，其左侧是一连串 0，从原数最右侧的“0”开始，一直到其右界，这一段距离所对应的掩码都是“1”。对于整数 $\boldsymbol{i}$ 最右侧的“0”来说，由表达式$\neg \boldsymbol{i}\ \&(\boldsymbol{i}+\mathbf{1})$ 构建出来的那个数，此位置上的值是“1”（参见 2.1 节）。要想递增某个未反转的整数 $\boldsymbol{x}$，可以把刚构建出来的那个数里的“1”向右传播，用转播后的值做掩码，和原 $\boldsymbol{x}$ 求“异或”。如果想“递增”位元反转后的整数，那么需要对掩码

⊖ 对以 −2 为基数（base−2）的数值也是如此，然而以 −1+i 为基数（base−1+i）的数值不具备此属性。

做镜像或位元反转。用 $m/2$ 除以 $\neg i \,\&\, (i+1)$，就可以对后者中值为“1”的那个位元做水平镜像。（这一步是算法的关键。）例如，对于可用4个二进制位容纳下的正整数来说，$m=2^4=16$，$m/2=8$，$8/1=8$，$8/2=4$，$8/4=2$，$8/8=1$。为了算出掩码，必须把商里值为“1”的位元向左传播，而此操作可由 m 减去商来完成。最后，对掩码与反转后的整数求“异或”，即可由原来的 i 值算出它下一个值 $i+1$ 反转位元后的那个整数了。

以8位二进制整数为例，$m=256$。设 $i=19$（二进制 **0001 0011**），则 $revi$ = 二进制 **1100 1000**。而 $\neg i \,\&\, (i+1)$ = 二进制 **0000 0100**（十进制的 **4**）。用 $m/2$ 除以 **4**，商是 **32**（二进制 **0010 0000**）。用 m 减去这个商，差的二进制值是 **1110 0000**。最后，将该掩码与 $revi$ 求“异或”，得出二进制 **0010 1000**，这也就是将十进制数 **20** 位元反转后的结果。

2. 注意到：

$$
\begin{aligned}
m_0 &= 2 \cdot \texttt{0x11111111} \\
m_1 &= \texttt{0xC} \cdot \texttt{0x01010101} \\
m_2 &= \texttt{0xF0} \cdot \texttt{0x00010001} \\
m_3 &= \texttt{0xFF00} \cdot \texttt{0x00000001}
\end{aligned}
$$

还观察到：

$$
\begin{aligned}
\texttt{0x11111111} &= \lfloor 2^{32}/15 \rfloor \\
\texttt{0x01010101} &= \lfloor 2^{32}/255 \rfloor \\
\texttt{0x00010001} &= \lfloor 2^{32}/(2^{16}-1) \rfloor \\
\texttt{0x00000001} &= \lfloor 2^{32}/(2^{32}-1) \rfloor
\end{aligned}
$$

因此可得出公式：

$$
\begin{aligned}
m_0 &= (2-1)2\lfloor 2^{32}/(2^4-1) \rfloor \\
m_1 &= (2^2-1)2^2\lfloor 2^{32}/(2^8-1) \rfloor \\
m_2 &= (2^4-1)2^4\lfloor 2^{32}/(2^{16}-1) \rfloor \\
m_3 &= (2^8-1)2^8\lfloor 2^{32}/(2^{32}-1) \rfloor
\end{aligned}
$$

综上可得：

$$m_k = (2^{2^k}-1)2^{2^k}\lfloor 2^W/(2^{2^{k+2}}-1) \rfloor$$

其中 W 是待洗牌字组的位元长度，它必须是2的幂。

3. 只需将下面两行：

```
s = s + b;
x = x >> 1;
```

改为

```
s = s + 1;
x = x >> b;
```

4. 只要是真正的LRU算法，就必须完整记录下“组”内 n 个缓存块的引用顺序。因为 n 个事物的全排共有 $n!$ 种，所以LRU算法至少需要用到 $\lceil \log_2 n! \rceil$ 个存储位。下表对比了此最小值与“引用矩阵法”所需的存储位数量。

结合度	理论最小值	引用矩阵法
2	1	1
4	5	6
8	16	28
16	45	120
32	118	496

第 8 章

1. 正如 8.3 节所述，若将带符号整数 x 与 y 作为操作数执行无符号乘法，则表示乘积的无符号整数是：

$$(x+2^{32}x_{31})(y+2^{32}y_{31})=xy+2^{32}(x_{31}y+y_{31}x)+2^{64}x_{31}y_{31}$$

其中 x_{31} 与 y_{31} 分别是 x 与 y 的符号位，不是 0 就是 1。由于此乘积与真实乘积 xy 之差为 2^{32} 的倍数，所以这两种乘积的低权重 32 位都一样。

2. 方法 1：由于计算机中也许有乘法指令能直接求出两个 32 位整数之积的低权重 32 位，所以，此时直接可得：

```
low = u*v;
```

方法 2：在 return 语句之前，插入

```
low = (w1 << 16) + (w0 & 0xFFFF);
```

方法 3：将 u1 * v0 与 u0 * v1 分别保存在临时变量 t1 和 t2 中。然后用下列代码计算：

```
low = ((t1 + t2) << 16) + w0;
```

方法 2 与 3 各需 3 条基本 RISC 指令，它们对 mulhs 函数及此函数的无符号版本都有效（而且可能比方法 1 快）。

3. 将 32 位操作数 u 与 v 划分成 16 位无符号分量 a、b、c、d，于是：

$$u=2^{16}a+b，且$$
$$v=2^{16}c+d$$

其中 $0\leqslant a, b, c, d\leqslant 2^{16}-1$。设

$$p=ac$$
$$q=bd$$
$$r=(-a+b)(c-d)$$

则可轻易推出：$uv=2^{32}p+2^{16}(r+p+q)+q$。

因为 $0\leqslant p, q\leqslant 2^{32}-2^{17}+1$，所以 p 与 q 都可以容纳于 32 位无符号整数中。然而，很容易算出：

$$-2^{32}+2^{17}-1\leqslant r\leqslant 2^{32}-2^{17}+1$$

因此 r 是个带符号的 33 位二进制数。用带符号的 64 位整数很容易表示它，只要把高

32位全部置0或置1即可。计算机乘法指令可以算出 r 的低权重32位，而高32位可由 $-a+b$ 及 $c-d$ 来确定。这两部分都是17位带符号整数。若其符号相反且均不是0，则 r 为负，因此其高32位全是“1”。若其符号相同或其中之一为0，则 r 非负，因此其高32位都是“0”。只需判断 r 的低32位是不是0，即可看出 $-a+b$ 与 $c-d$ 中是否有一个为0。如果 r 的低32位值是0，则两个因数中至少有一个是0，因为 $|r|<2^{32}$。上述推导过程可归结为下列函数，它能算出 u 与 v 之积的高权重32位。

```
unsigned mulhu(unsigned u, unsigned v) {
   unsigned a, b, c, d, p, q, rlow, rhigh;

   a = u >> 16;  b = u & 0xFFFF;
   c = v >> 16;  d = v & 0xFFFF;

   p = a*c;
   q = b*d;
   rlow = (-a + b)*(c - d);
   rhigh = (int)((-a + b)^(c - d)) >> 31;
   if (rlow == 0) rhigh = 0;     // Correction.
   q = q + (q >> 16);    // Overflow cannot occur here.
   rlow = rlow + p;
   if (rlow < p) rhigh = rhigh + 1;
   rlow = rlow + q;
   if (rlow < q) rhigh = rhigh + 1;

   return p + (rlow >> 16) + (rhigh << 16);
}
```

计算完 p、q、rlow、rhigh 之后，函数所做加法如下：

```
              |......p.......|
|....rhigh.....||.....rlow......|
               |.......p.......|
               |.......q.......|
                       |.......q.......|
```

“if (rlow<p) rhigh = rhigh + 1”这条语句的意思是，如果前一条语句所执行的 rlow 加 p 进位了，则为 rhigh 加1。

乘积的低权重32位由下式给出，把它插在“修正”(correction) 步骤的后面即可：

```
q + ((p + q + rlow) << 16)
```

下面列出无分支版。

```
unsigned mulhu(unsigned u, unsigned v) {
   unsigned a, b, c, d, p, q, x, y, rlow, rhigh, t;

   a = u >> 16;  b = u & 0xFFFF;
   c = v >> 16;  d = v & 0xFFFF;

   p = a*c;
   q = b*d;
   x = -a + b;
   y = c - d;
   rlow = x*y;
   rhigh = (x ^ y) & (rlow | -rlow);
   rhigh = (int)rhigh >> 31;
```

```
    q = q + (q >> 16);    // Overflow cannot occur here.
    t = (rlow & 0xFFFF) + (p & 0xFFFF) + (q & 0xFFFF);
    p += (t >> 16) + (rlow >> 16) + (p >> 16) + (q >> 16);
    p += (rhigh << 16);
    return p;
}
```

这些函数的开销都比 8.2 节中图 8.2 所列函数大，而且只有当计算机的乘法指令比当今绝大部分计算机慢时，才能体现其优势。在高精度计算（"bignum" arithmetic，"大数"算术，也就是以多字整数为操作数的算术）中，乘法耗时远高于计算类似大小的两整数之和所费的时间。对于这种应用场景来说，有一种算法叫做"Karatsuba 乘法"[⊖]（Karatsuba multiplication）[Karat]，它递归执行本题所讲的这种只使 3 次乘法的方案（three-multiplication scheme），只要操作数足够大，它就比使用 4 次乘法的那个方案（four-multiplication scheme）更快。实际上，一般所说的 Karatsuba 乘法使用如下算式：

$$p=ac$$
$$q=bd$$
$$r=(a+b)(c+d)$$
$$uv=2^{32}\ p+2^{16}(r-p-q)+q$$

对本题来说，这种办法效果不好，因为 r 可能大得快接近 2^{34} 了，似乎没有简单的办法能算出 34 位二进制数 r 的那两个高权重位。

带符号版的函数有溢出问题。可先用无符号版，然后根据 8.3 节所述方法修正计算结果。

第 9 章

1. 设 $x=x_0+\delta$，其中 x_0 为整数且 $0\leqslant\delta<1$。根据"向上取整函数"（ceiling function，天花板函数）的定义，函数值是下一个大于等于其参数的整数，因此可得$\lceil -x\rceil=\lceil -x_0-\delta\rceil=-x_0$。于是$-\lceil -x\rceil=x_0$，而这也就是$\lfloor x\rfloor$的值。
2. 用 n/d 表示带符号整数的"向 0 取整式除法"之商，我们必须分情况计算"向下取整式除法"之商。

$$\text{当 } n\geqslant 0,\ d>0 \text{ 时，计算 } n/d$$
$$\text{当 } n<0,\ d>0 \text{ 时，计算 } n/d-1$$
$$\text{当 } n\geqslant 0,\ d<0 \text{ 时，计算 } n/d$$
$$\text{当 } n<0,\ d<0 \text{ 时，计算 } n/d+1$$

（$d=0$ 时，结果无所谓。）上述 4 种情况可以统一用 $n/d+c$ 算出，其中

⊖ 此算法由俄罗斯数学家 Anatolii Alexeevitch Karatsuba（1937—2008）发明，详情可参见：http://en.wikipedia.org/wiki/Karatsuba_algorithm。——译者注

$$c=((n\overset{s}{\gg}31\oplus(d\overset{s}{\gg}31)-(d)\overset{s}{\gg}31)$$

需要4条指令来计算 c（将两个 $d\overset{s}{\gg}31$ 合并）。另外一种计算 c 的办法也需4条指令，不过移位操作是无符号的：

$$c=(d\overset{u}{\gg}31)-((n\oplus d)\overset{u}{\gg}31)$$

如果计算机有“模32”移位指令，那么可用3条指令计算 c：

$$c=(n\overset{s}{\gg}31)\overset{u}{\gg}(d\overset{s}{\gg}31)$$

现在来看余数怎样算。用“向0取整式除法”将带符号整数 n 除以带符号整数 d，把余数记为 $\mathrm{rem}(n,d)$。据此，可以分情况算出“向下取整式除法”的余数。

当 $n\geqslant 0$，$d>0$ 时，计算 $\mathrm{rem}(n,d)$
当 $n<0$，$d>0$ 时，计算 $\mathrm{rem}(n,d)+d$
当 $n\geqslant 0$，$d<0$ 时，计算 $\mathrm{rem}(n,d)$
当 $n<0$，$d<0$ 时，计算 $\mathrm{rem}(n,d)-d$

也就是说，要给 $\mathrm{rem}(n,d)$ 加上一个数，这个数要么是0，要么是 d 的绝对值。可用如下方法算出：

$$|d|=(d\oplus(d\overset{s}{\gg}31))-(d\overset{s}{\gg}31)$$

$$c=|d|\&(n\overset{s}{\gg}31)$$

需要用5条指令计算 c。如果计算机有“模32”移位指令，并且我们以乘法指令来计算 c，那么用4条指令就够了（细节从略）。

3. 要计算“向下取整式除法”的商，只需在被除数与除数的正负号相反时，把“向0取整式除法”的商减1即可：

$$n/d=((n\oplus d)\overset{u}{\gg}31)$$

要计算余数，只需在被除数与除数的正负号相反时，把除数加到“向0取整式除法”的余数上即可：

$$\mathrm{rem}(n,d)+(((n\oplus d)\overset{s}{\gg}31)\&d)$$

4. 大家一般会想到的方法很可能是计算 $\lfloor(\boldsymbol{n}+\boldsymbol{d}-\mathbf{1})/\boldsymbol{d}\rfloor$ 的值。问题在于，$\boldsymbol{n}+\boldsymbol{d}-\mathbf{1}$ 可能会溢出。（比如在4位计算机上用此方法计算 $\lceil 12/5\rceil$。）

另一种常规的方法是用计算机的除法指令来计算 $\lfloor \boldsymbol{n}/\boldsymbol{d}\rfloor$，将其值设为 $\boldsymbol{q}$，然后按 $\boldsymbol{r}=\boldsymbol{n}-\boldsymbol{qd}$ 来计算余数 $\boldsymbol{r}$，若 $\boldsymbol{r}$ 非0，则给 $\boldsymbol{q}$ 加 $\mathbf{1}$。（或者在 $\boldsymbol{n}\neq\boldsymbol{qd}$ 时给 $\boldsymbol{q}$ 加 $\mathbf{1}$。）对于所有的 $\boldsymbol{n}$、$\boldsymbol{d}(\boldsymbol{d}\neq\mathbf{0})$ 来说，这样算出来的结果都正确，只是由于要计算乘法、减法，而且可能还要为 $\boldsymbol{q}$ 加 $\mathbf{1}$，所以此算法的开销有些大。从另一方面来看，如果计算机的除法指令能够顺便求出余数，那么这个方法也还不错，尤其是当CPU能高效地算出 $\boldsymbol{q}=\boldsymbol{q}+(\boldsymbol{r}\neq\mathbf{0})$ 时更是如此。

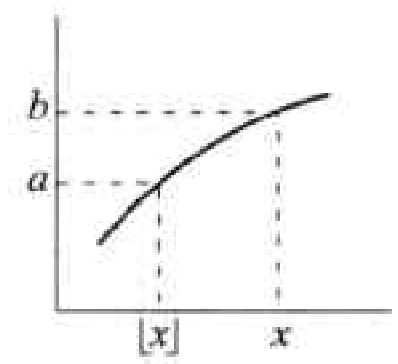
除了上述方法外，还有一种做法，就是计算 $\boldsymbol{q}=\lfloor(\boldsymbol{n}-\mathbf{1})/\boldsymbol{d}\rfloor+\mathbf{1}$。可惜在 $\boldsymbol{n}=\mathbf{0}$ 时该方法

无效。如果计算机能够轻易算出“$x \neq 0$”这个谓词的值，那么就能修复此问题。比方说，可以用比较指令把谓词化为整数 **0** 或 **1**，并将其存于目标寄存器中（参见 2.12 节）。这样一来，就可以按如下方式计算了：

$$\boldsymbol{c} \leftarrow (\boldsymbol{x} \neq \mathbf{0})$$
$$\boldsymbol{q} \leftarrow \lfloor (\boldsymbol{n}-\boldsymbol{c})/\boldsymbol{d} \rfloor + \boldsymbol{c}$$

最后要说的是，用 $\boldsymbol{q}=\lfloor(\boldsymbol{n}-\mathbf{1})/\boldsymbol{d}\rfloor+\mathbf{1}$ 来计算也可以，只要能在 $\boldsymbol{n}=\mathbf{0}$ 时把结果设为 **0** 就好。可通过“条件移动指令”（conditional move instruction）或“选取指令”（select instruction）等方式来做到这一点。

5. 如下图所示，设 $f(\lfloor x \rfloor)=a$ 且 $f(x)=b$。



若 b 为整数，则根据属性（c）可知 x 也是整数，于是 $\lfloor x \rfloor = x$，此时无需继续证明。下述证明过程中假设 b 不是整数，而 a 可以是整数，也可以不是。

不可能找到满足关系式 $a<k\leqslant b$ 的整数 k，因为假如有这样的整数 k，那么根据属性（a）、（b）、（c）可知：$\lfloor x \rfloor$ 与 x 之间会出现一个整数。而这是不可能的，因此，$\lfloor a \rfloor = \lfloor b \rfloor$，也就是说 $\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$。

利用此规律，可举出下面几个例子，它们在 a 和 b 是整数时成立：

$$\left\lfloor \frac{\lfloor x \rfloor + a}{b} \right\rfloor = \left\lfloor \frac{x+a}{b} \right\rfloor$$
$$\lfloor \sqrt{\lfloor x \rfloor} \rfloor = \lfloor \sqrt{x} \rfloor$$
$$\lfloor \log_2(\lfloor x \rfloor) \rfloor = \lfloor \log_2(x) \rfloor$$

同理还可证明：若 $f(x)$ 满足属性（a）、（b）、（c），则

$$\lceil f(\lceil x \rceil) \rceil = \lceil f(x) \rceil$$

第 10 章

1. 以下证明（a）操作正确。如果除数是偶数，那么被除数最低位不影响商（假设按“向下取整式除法”来算），此位若为“1”，则余数为奇数。如果清空此位，则除法的余数也是偶数。因此对于偶除数 d 来说，余数最多是 $d-2$。当余数可取的最大值降低了之后，乘数 m 最大值所占的位数也由原来的（$W+1$）位变成了现在的 W 位（这样的话，就不用 shrxi 指令了）。实际上，如果除数以 z 个 0 位结尾，也就是说它是 2^z 的倍数（其中 $z\geqslant 0$），那么除法就能简化。在此情况下，把被除数的最低 z 位清空不影响商，而且清空了这些位后，最大余数就是 $d-2^z$ 了。

按照 10.9 节的思路推导，把余数最大值换为 $d-2^z$，于是可得 $n_c=2^W-\mathrm{rem}(2^W, d)-2^z$，不等式（24a）变成：

$$2^W-d \leqslant n_c \leqslant 2^W-2^z$$

不等式（25）变成

$$\frac{2^p}{d} \leqslant m < \frac{2^p}{d} \frac{n_c + 2^z}{n_c}$$

等式（26）不变，不等式（27）变为

$$2^p > \frac{n_c}{2^z}(d - 1 - \text{rem}(2^p - 1, d)) \tag{27'}$$

不等式（28）变为

$$1 \leqslant 2^p \leqslant \frac{2n_c}{2^z}(d-1) + 1$$

如果不强令 p 等于 W，则将上述不等式合并可得：

$$\frac{1}{d} \leqslant m < \frac{2n_c(d-1) + 2^z}{2^z d} \frac{n_c + 2^z}{n_c}$$

$$1 \leqslant m < \frac{2d - 2 + 2^z/n_c}{2^z d}(n_c + 2^z)$$

$$1 \leqslant m < \frac{2}{2^z}(n_c + 2^z) \leqslant \frac{2}{2^z} 2^W$$

因此，如果 $z \geqslant 1$，那么 $m < 2^W$，也就是说 m 能存放在 W 位的字组中。在强令 p 等于 W 的情况下亦有相同结果。

给定某个除数，现在计算乘数 m 的方法是：先按照上面的式子算出 n_c，再寻找符合 $p \geqslant W$ 且能令（27′）式成立的最小 p 值，然后根据（26）式算出 m。举例来说，在 $d=14$ 且 $W=32$ 时，可得 $n_c = 2^{32} - \text{rem}(2^{32}, 14) - 2 = \text{0xFFFF FFFA}$。反复运用（27′）可得 $p=35$，再根据（26）式算出 $m = (2^{35} + 14 - 1 - 3)/14 = \text{0x9249 2493}$。因此，除以 14 的代码就是：

```
ins    n,R0,0,1         Clear low-order bit of n.
li     M,0x92492493     Load magic number.
mulhu  q,M,n            q = floor(M*n/2**32).
shri   q,q,3            q = q/8.
```

以下证明（b）操作正确。与（a）一样，如果除数是 2^z 的倍数，那么被除数最低 z 位不影响商。因此，若清除被除数最低 z 位，并将原除数除以 2^z，则当前被除数与当前除数之商与原来相同。（原除数除以 2^z 这一操作可在编译时执行。）

因为修改过的 n_c 和 d 均小于 2^{W-z}，所以（24a）式变成

$$2^{W-z} - d \leqslant n_c \leqslant 2^{W-z} - 1$$

等式（26）与不等式（27）不变，然而现在要把修改过的 n_c 与 d 代入其中。此处省去证明乘数小于 2^W 的过程，给出 $d=14$ 且 $W=32$ 时的范例代码。在相关算式中，我们设 $d=7$。因而可得 $n_c = 2^{31} - \text{rem}(2^{31}, 7) - 1 = \text{0x7FFF FFFF}$。反复运用（27）式，可得 $p=34$，再根据（26）式可知 $m = (2^{34} + 5)/7 = \text{0x9249 2493}$，因此，除以 14 的代码就是：

```
shri   n,n,1            Halve the dividend.
li     M,0x92492493     Load magic number.
mulhu  q,M,n            q = floor(M*n/2**32).
shri   q,q,2            q = q/4.
```

这并不是说只要除数为偶数就一定要用上面两种方法。例如，在除数为 10、12、18、22 时，最好采用本书正文中的方法，因为在这些情况下，本来就不需要用指令清除被除数的低权重位，也不需要把被除数右移。反之，应该先采用 10.11.3 节的图 10.3，然后，若发现“add 指示器”是 1 且除数为偶数，再于上述两技巧中择一，这样的话，在大部分计算机上都能编出更好的代码。在小于 100 的除数中，值得运用上述技巧的是：14、28、38、42、54、56、62、70、74、76、78、84、90。

那么（a）和（b）哪个好呢？试验表明，从所需指令数上看，（b）方法更优，其指令数似乎总是和（a）相同或比它少 1。然而某些情况下，（a）、（b）指令数相同，但（a）生成的乘数较小。下面列出了一些典型情况。“Book” method（“教科书”方法）指的就是图 10.3。此处假设计算机的“同立即数求与”（and immediate）指令会对立即数字段的最高有效位做符号扩展（在基本 RISC 架构的计算机上可使用 insert 指令来实现符号扩展功能）。

$d=6$

Book	(a)	(b)

`li M,0xaaaaaaab`
`mulhu q,M,n`
`shri q,q,2`
	`andi n,n,-2`
	`li M,0x2aaaaaab`
	`mulhu q,M,n`
		`shri n,n,1`
		`li M,0x55555556`
		`mulhu q,M,n`

$d=28$

Book	(a)	(b)

`li M,0x24924925`
`mulhu q,M,n`
`add q,q,n`
`shrxi q,q,5`
	`andi n,n,-4`
	`li M,0x24924925`
	`mulhu q,M,n`
	`shri q,q,2`
		`shri n,n,2`
		`li M,0x24924925`
		`mulhu q,M,n`

这些技巧对带符号除法用处不大。在带符号的情况下，最优代码与最差代码只差两条指令（可以比较 10.3 节除以 3 和除以 7 的代码）。为了令方法（a）在带符号情况下正确，必须在被除数为负奇数时加 1，在其为非负的奇数时减 1，这样需要的指令数大于 2。为了令方法（b）正确应对带符号除法，必须把被除数除以 2，而这需要 3 条基本 RISC 指令（参见 10.1 节），所以与书中正文所述方法相比，此方法也无改进。

2. Python 语言代码如下。

```
def magicg(nmax, d):
   nc = (nmax//d)*d - 1
   nbits = int(log(nmax, 2)) + 1
   for p in range(0, 2*nbits - 1):
      if 2**p > nc*(d - (2**p)%d):
         m = (2**p + d - (2**p)%d)//d
         return (m, p)
   print "Can't find p, something is wrong."
   sys.exit(1)
```

3. 因为 $81=3^4$，所以需要先求出 d 对于模 3 的“乘法逆元素”，以此作为运用“牛顿法”所需的初始值。由于 $1\cdot 1\equiv 1 \pmod 3$ 且 $2\cdot 2\equiv 1 \pmod 3$，因此这个乘法逆元素也就

是 d 除以 3 的余数（若余数为 0，则 d 没有"乘法逆元素"）。当 $d=146$ 时，计算过程如下。

$$x_0 = 146 \bmod 3 = 2$$
$$x_1 = 2(2-146\cdot 2) = -580 \equiv 68 \pmod{81}$$
$$x_2 = 68(2-146\cdot 68) = 674\ 968 \equiv 5 \pmod{81}$$
$$x_3 = 5(2-146\cdot 5) = -3640 \equiv 5 \pmod{81}$$

由于算出的值固定不变了，所以 146 对于模 81 的乘法逆元素就是 5。验算：$146\cdot 5=730\equiv 1 \pmod{81}$。实际上我们预先知道：仅需两次迭代就够了。

第 11 章

1. 两者相等。尽管经过两次截取，结果仍然正确。假设$\lfloor\sqrt{x}\rfloor=a$。按照此操作的定义，由于 a 是整数，所以有 $a^2\leqslant x$ 且 $(a+1)^2>x$。

   设$\lfloor\sqrt{a}\rfloor=b$。于是 $b^2\leqslant a$ 且 $(b+1)^2>a$。因为 $b^4\leqslant a^2$ 且 $a^2\leqslant x$，所以 $b^4\leqslant x$。

   因为 $(b+1)^2>a$，也就是说 $(b+1)^2\geqslant a+1$，所以 $(b+1)^4\geqslant(a+1)^2$。又因为 $(a+1)^2>x$，所以 $(b+1)^4>x$。因此，b 是 x 的整数 4 次方根。

   直接利用第 9 章习题 5 的结论，此题会容易很多。

2. 直观的实现代码如下。

```
int icbrt64(unsigned long long x) {
   int s;
   unsigned long long y, b, bs;

   y = 0;
   for (s = 63; s >= 0; s = s - 3) {
      y = 2*y;
      b = 3*y*(y + 1) + 1;
      bs = b << s;
      if (x >= bs && b == (bs >> s)) {
         x = x - bs;
         y = y + 1;
      }
   }
   return y;
}
```

   只有在第二次循环迭代时 b 才会溢出（上述代码中的 bs）。因此，另一种处理溢出的办法是把前两轮循环迭代展开，然后从 s=57 开始以循环方式执行，并删掉其中的"&& b== (bs≫s)"。

   仔细观察可知，前两轮循环迭代的效果是：

   如果 $x\geqslant 2^{63}$，就将 x 设为 $x-2^{63}$，并将 y 设为 2。

   如果 $2^{60}\leqslant x<2^{63}$，就将 x 设为 $x-2^{60}$，并将 y 设为 1。

   如果 $x<2^{60}$，将 y 设为 0（x 不变）。

   因此，算法开头的代码可改为

```
y = 0;
if (x >= 0x1000000000000000LL) {
   if (x >= 0x8000000000000000LL) {
      x = x - 0x8000000000000000LL;
      y = 2;
   } else {
      x = x - 0x1000000000000000LL;
      y = 1;
   }
}

for (s = 57; s >= 0; s = s - 3) {
   ...
```

正如刚才所说，“&&b==(bs≫s)”可以从循环中删掉。

3. 需要 6 次乘法 [Knu2]。基于 $x^{23}=x^{16}\cdot x^4\cdot x^2\cdot x$ 的二进制分解法需要 7 次。将 x^{23} 分解为 $(x^{11})^2\cdot x$ 或 $((x^5)^2\cdot x)^2\cdot x$ 也需要 7 次。按照 x^2，x^3，x^5，x^{10}，x^{13}，x^{23} 的顺序依次算出 x 的各次幂，x^2 是 x 与 x 之积，x^3 是 x^2 与 x 之积，其他各项都是它前面某两项的乘积，这样算下来只需 6 次乘法。

4. (a) 将 x 向下取整为 2 的整数幂。(b) 将 x 向上取整为 2 的整数幂。(在这两种情况下，若 x 本身已是 2 的整数幂，则函数值就是 x)。

第 12 章

1. 若 $\boldsymbol{B}$ 为二进制数，而 $\boldsymbol{N}$ 为负二进制数，则有：

$$\boldsymbol{B}\leftarrow \mathbf{0x5555\ 5555}-(\boldsymbol{N}\oplus\mathbf{0x5555\ 5555})$$

$$\boldsymbol{N}\leftarrow(\mathbf{0x5555\ 5555}-\boldsymbol{B})\oplus\mathbf{0x5555\ 5555}$$

2. 有种简单办法：把负二进制数 $\boldsymbol{x}$ 转为二进制，加 1，再转回负二进制。套用 Schroeppel 公式并化简，可得：

$$((\boldsymbol{x}\oplus\mathbf{0xAAAA\ AAAA})+\mathbf{1})\oplus\mathbf{0xAAAA\ AAAA}\ \text{或}$$

$$((\boldsymbol{x}\oplus\mathbf{0x5555\ 5555})-\mathbf{1})\oplus\mathbf{0x5555\ 5555}$$

3. 与习题 2 相仿，可将负二进制数 $\boldsymbol{x}$ 转为二进制，同 0xFFFF FFF0 做“与”运算，再转回负二进制。这样需要 5 次操作。然而，在下述两公式中任选其一，则只需 4 次[⊖]。

$$(((\boldsymbol{x}\oplus\mathbf{0xAAAA\ AAAA})-\mathbf{10})\oplus\mathbf{0xAAAA\ AAAA})\ \&-16$$

$$(((\boldsymbol{x}\oplus\mathbf{0x5555\ 5555})+\mathbf{10})\oplus\mathbf{0x5555\ 5555})\ \&-16$$

下列公式把负二进制数向上调整为下一个 16 的倍数。

$$(((\boldsymbol{x}\oplus\mathbf{0xAAAA\ AAAA})+\mathbf{5})\oplus\mathbf{0xAAAA\ AAAA})\ \&-16$$

$$(((\boldsymbol{x}\oplus\mathbf{0x5555\ 5555})-\mathbf{5})\oplus\mathbf{0x5555\ 5555})\ \&-16$$

如果想上调或下调至 2 的其他次幂，那么也有与之相似的公式。

4. 由于 Python 语言支持复数，所以用其编码很容易。

⊖ 这些公式是用“Aha!”程序找出来的，该程序使用穷举法来搜寻最优表达式。

```
import sys
import cmath

num = sys.argv[1:]
if len(num) == 0:
   print "Converts a base -1 + 1j number, given in decimal"
   print "or hex, to the form a + bj, with a, b real."
   sys.exit()
num = eval(num[0])
r = 0
weight = 1
while num > 0:
   if num & 1:
      r = r + weight;
   weight = (-1 + 1j)*weight
   num = num >> 1;
print 'r =', r
```

5. 要把$-1+i$进制数转为其相反数，可以将其从0中减去，也可以把它和-1（也就是11101）相乘，这两种方式都要使用$-1+i$进制下的算术规则。

为了提取$-1+i$进制数x的实部，需要把虚部的相反数与其自身相加。把x中每4位编为一组，从右侧（低权重位）开始处理。每一组的4个数位从右至左分别叫做0号、1号、2号、3号。于是：

若1号位是1，则为当前小组加上$-i$（0111）。

若2号位是1，则为当前小组加上$2i$（1110110）。

若3号位是1，则为当前小组加上$-2i$（0100）。

1号位的权重是$-1+i$，所以为其加上$-i$也就等于消去其虚部。对2号位和3号位的处理也基于此思路。由于0号位没有虚部，故无须处理。因为$-1+i$进制的10 000就是十进制的-4，所以每个4位小组的权重都是其右侧那一组的-4倍。于是，x中第n位的权重等于一个实数（此数就是-4）乘以第$n-4$位的权重。

下面举例说明如何提取$-1+i$进制数1 0110 1101的实部。

1 0110 1101	x
111 0100	给第2位加上$2i$
0100	给第3位加上$-2i$
0111	给第5位加上$-i$（-4）
111 0100	给第6位加上$2i$（-4）
1100 1101 1101	求和结果

读者可以验证一下：当x为$23+4i$时，求和结果是23。在计算加法的过程中会产生许多进位，上面所演示的计算过程没有将其写出。其实有捷径可循：若2号位与3号位都是1，则无需对其执行加法，因为这相当于加上$2i$之后再加$-2i$。如果某小组以11结尾，则直接丢弃这些数位，因为它们表示纯虚数i。同理，2号位也可以直接丢弃，因为其权重是纯虚数$-2i$。

完全套用上述思路就能通过此类技巧构建出一种算法：此算法可把每个包含4位数的

小组独立转译为该组的实部。某些情况下可能有进位，把进位加到转译后的数值上就好。为了演示此方法，我们把每个小组内的 4 个数位用一个十六进制数表示。转译规则如下。

0⇒0	4⇒0	8⇒C	C⇒C
1⇒1	5⇒1	9⇒D	D⇒6
2⇒1D	6⇒1D	A⇒1	E⇒1
3⇒0	7⇒0	B⇒C	F⇒C

在 $-1+i$ 进制下，数位 2 与 6 所代表的数，其实部都是 -1，而 -1 这个值在本题所用的这种 $-1+i$ 进制里写作 1D。对于这两种数位值来说，把原数位改为 D，然后带出进位 1 即可。可用 $-1+i$ 进制下的基本加法规则把进位加到下一位上，不过要是手算的话，还有个更方便的办法。根据上述转译表可知，转译之后，只可能出现 4 种数位：0、1、C、D。下表左侧一栏列出了这些数位同 1 相加的规则。

0+1=1	0+1D=1D
1+1=C	1+1D=0
C+1=D	C+1D=1
D+1=1D0	D+1D=C

给 D 加 1，会产生值为 1D 的进位（因为 $3+1=4$）。我们把进位中的那两个数位一并加到同一个位置中。上表右侧一栏列出了如何处理值为 1D 的进位。在做加法时，某个位置上可能会出现由前一个位置所带来的两个进位，一个是 1，另一个是 1D（前一个进位是在转译过程中产生的，而后一个进位是在加法过程中产生的）。在这种情况下，两个进位互相抵消，因为 $-1+i$ 进制下的 1D 其实就是 -1。在同一个位置上，不可能出现两个值为 1 或两个值为 1D 的进位。

下例演示了如何用这种方法将实部从写成十六进制数位的 $-1+i$ 进制数 EA26 中提取出来。

```
EA26  x
 11   转译过程产生的进位
11DD  x 转译之后的十六进制数位
-------
110D  求和结果
```

读者可以验证，当 x 为 $-45+21i$ 时，求和结果是 -45。

顺便说一下，一个使用十六进制数位的 $-1+i$ 进制数，当且仅当其每个数位都是 0、1、C 或 D 时，其值才会是实数。

想提取 x 的虚部，当然可以先提取实部，然后把它从 x 里减去。也可以直接采用“捷径”法，按照下表将每一个写成十六进制的数位转译为与之对应的纯虚部。

0⇒0	4⇒4	8⇒74	C⇒0
1⇒0	5⇒4	9⇒74	D⇒0

2⇒3	6⇒7	A⇒77	E⇒3
3⇒3	7⇒7	B⇒77	F⇒3

由上表可知，进位有可能是 7，于是我们需要知道如何按加法规则把 7 加到 0、3、4、7 这 4 种可能出现的数位之上。下表左栏列出了这 4 条规则。

0+7=7	0+3=3
3+7=0	3+3=74
4+7=33	4+3=7
7+7=4	7+3=0

按上表左栏规则处理进位时，又可能出现值为 3 的进位，因而还需按照右边一栏里的规则处理此进位。

下例演示了如何用这种方法将虚部从写为十六进制数位的 −1+i 进制数 568A 中提取出来。

```
568A  x
  77  转译过程产生的进位
4747  x 转译之后的十六进制数位
----
4737  求和结果
```

读者可以验证一下，当 x 为 −87+107i 时，求和结果是 107i。

使用十六进制数位的 −1+i 进制数，当且仅当其每个数位都是 0、3、4 或 7 时，其值才是纯虚数。

想把一个数变为其共轭复数，可以连减两次虚部。也可以采用上面提到的转译表，不过这次更复杂了，一来因为进位更多，二来因为转译过的数位不像原来那样仅局限于 4 种，而是有可能包含全部 16 种十六进制数位。转译表如下。

0⇒0	4⇒74	8⇒38	C⇒C
1⇒1	5⇒75	9⇒39	D⇒D
2⇒6	6⇒2	A⇒3E	E⇒3A
3⇒7	7⇒3	B⇒3F	F⇒3B

产生的进位可根据 −1+i 进制下的算术规则来处理，也可构建一张表格，每次根据此表求出一个十六进制数位与一个进位之和。由于进位可能会和全部 16 种十六进制数位相加，所以这张表比前面几张要大。

第 13 章

1. 证明方法 1 概要：根据二进制反射格雷码的构建方式，显然可知此论述成立。

   证明方法 2 概要：根据公式 $G(\boldsymbol{x})=\boldsymbol{x}\oplus(\boldsymbol{x}\overset{u}{\gg}1)$ 可知，如果第 i 位是 0，而其左侧位元是 1，或第 i 位是 1，而其左侧位元是 0，那么 $G(\boldsymbol{x})$ 的第 i 位就是 1，否则就是 0。若 $\boldsymbol{x}$ 为偶数，则"$G(\boldsymbol{x})$ 的某位是 1"这一现象会出现偶数次，而 $\boldsymbol{x}$ 若为奇数，则上述现象

会出现奇数次。

证明方法 3 概要：对 $\boldsymbol{x}$ 的长度运用数学归纳法。根据上述公式可知，字长为 1 时，也就是 $\boldsymbol{x}$ 为 0 或 1 时，此论述显然成立。设 $\boldsymbol{x}$ 是字长为 n 的二进制数，那么数学归纳法的归纳假设就是：假设待证明的命题对字长为 n 的 $\boldsymbol{x}$ 成立。现在要证明命题对字长为 $n+1$的新 $\boldsymbol{x}$ 也成立。若给原 $\boldsymbol{x}$ 左侧添 0，则新的 $G(\boldsymbol{x})$ 最左侧位元也是 0，而 $G(\boldsymbol{x})$ 的其余位元不变。若给原 $\boldsymbol{x}$ 左侧添 1，则新的 $G(\boldsymbol{x})$ 最左侧位元也是 1，而现在处于最高有效位右侧的那个位元则是其旧值的补值。$G(\boldsymbol{x})$ 其余位元保持不变。因此，在新的 $G(\boldsymbol{x})$ 中，值为 1 的位元数要么比原来多两个，要么不变。

因此，如果想制作一个随机数生成器，令其生成的数中含有偶数（或奇数）个 1，那么，可先用另一个生成器产生一个均匀分布的随机整数，然后将其最低有效位设成 0（或 1），最后转回格雷码即可 [Arndt]。

2. (a) 由于 STGC 的每一列都可由第 1 列垂直旋转而得出，所以此命题显然成立。

(b) 没有这种格雷码。把 $n=3$ 时的每一套格雷码都列出来，就不难证明此论述了。为了不失一般性，可以从下列格雷码开始：

```
000
001
011
......
```

由上述列表开始，对其中的整列求补并重排各列，即可推算出每一套可能出现的格雷码。由 (a) 和 (b) 可得出一条推论：当 $n=3$ 时，没有包含 8 个代码字的 STGC。

3. 选出长度为 3 的前 5 个二进制反射格雷码，将其对折并补位，即可得出下面这套格雷码。

```
0000
0001
0011
0010
0110
1110
1010
1011
1001
1000
```

还有一种办法是：先把十进制数化为“超 3-BCD 码”[⊖]，然后再转为格雷码。这样得出的格雷码也是循环格雷码。用超 3-BCD 码来编码十进制数位有个好处，那就是：相加会产生进位的两个十进制数位如果都化成超 3-BCD 码，那么，这两个新的代码字加起来仍会产生进位。

⊖ BCD 是 Binary-Coded Decimal 的简称，中文又称 BCD 码、二—十进制编码、二进码十进数，是一种以二进制位元来表示十进制数的编码形式。存在各种变体，“超 3-BCD 码”(excess 3 binary coded decimal，又称“余 3-BCD 码”）就是其中一种。此编码方式先将十进制数加 3，然后再化为二进制，如十进制数 9，加上 3 是 12，化为二进制是 1100，所以十进制数 9 的超 3-BCD 码就是 1100。详情参见：http://zh.wikipedia.org/wiki/二进码十进数。——译者注

超3格雷码

十进制数位	超3-BCD码	与左栏等价的格雷码
0	0011	0010
1	0100	0110
2	0101	0111
3	0110	0101
4	0111	0100
5	1000	1100
6	1001	1101
7	1010	1111
8	1011	1110
9	1100	1010

4. 按照编订反射格雷码所依据的原则，很容易就能设计出一套“混合进制”（“mixed base”）格雷码。若某数的质因子分解式为 $2^{e_1}3^{e_2}5^{e_3}\cdots$，则这套格雷码从左至右每一列所对应的底应为 e_1+1、e_2+1、e_3+1、…，例如，$72=2^3\cdot 3^2$，下表左栏列出了“四进制—三进制”混合格雷码，右栏列出了每个代码字所对应的因子，这些因子都能整除72。

```
00   1
01   3
02   9
12  18
11   6
10   2
20   4
21  12
22  36
32  72
31  24
30   8
```

显然，上述每个因子都是前一个因子与某质因子之积或商。

还有一种更简单的方法：将所有质因子组成一个集合，然后用一套二进制格雷码来决定当前这一步应该选取此集合的哪一个子集，反复在这些子集上迭代即可，每次的子集和上次相比，其中不是多了一个质因子，就是少了一个质因子。

第14章

1. 根据本章正文所述，信息多项式 M 与生成多项式 G 满足关系式：$Mx^r=QG+R$，其中 R 为校验和多项式。设 M' 为另一个信息多项式，其 x^e 项与 M 不同。（也就是说，在序号为 e 的位置上，两条信息的二进制值不同。）那么，$M'=M+x^e$，而且

$$M'x^r=(M+x^e)x^r=Mx^r+x^{e+r}=QG+R+x^{e+r}$$

因为能整除 x^{e+r} 的项必须是 x^n 的形式，而 G 却至少有两项不可能是 x^n 的形式，所以说，x^{e+r} 这一项无法被 G 整除。因此，$M'x^r$ 除以 G 之后的余数肯定和 R 不同，于是就能检测出错误了。

2. 主循环的代码可改写如下，其中变量 word 是 unsigned int（无符号整数）类型［Danne］。

```
crc = 0xFFFFFFFF;
while (((word = *(unsigned int *)message) & 0xFF) != 0) {
   crc = crc ^ word;
   crc = (crc >> 8) ^ table[crc & 0xFF];
   crc = (crc >> 8) ^ table[crc & 0xFF];
   crc = (crc >> 8) ^ table[crc & 0xFF];
   crc = (crc >> 8) ^ table[crc & 0xFF];
   message = message + 4;
}
```

与 14.3.2 节的图 14.7 相比，上述代码在处理 message 中的每个字组时，节省了 3 条字节加载指令与 3 条异或指令。此外，这段代码执行的循环控制指令也更少。

第 15 章

1. 构建出的表格应与 15.2 节的表 15.1 相似，只是最右侧一栏与奇数列都删去了。
2. 如果采用第一种做法，那么在某个校验位出错时，接收者无从得知这一状况，于是其会根据错误的校验位来“修正”信息位，这种修正操作本身就不对。

   如果用第二种做法，那么在某个校验位出错时，错误识别码将可能会是 100…0，010…0，001…0，…，000…1 等 k 种情况。因此，k 值必须足够大，以便将前述 k 种情况编码在内，同时还要再有 m 种值，以指明信息位中单一位元出错的情况，另外还要用一个值表示“传输无误”的情况。由此可见，k 的值必须符合汉明规则。

   依照此思路，有一种正确的做法：再使用一个奇偶校验位来表示原来那 k 个校验位的总体奇偶性，而那 k 个校验位则表示信息位里有一位传错的各种情况（并且要指明其位置）以及传输无误的情况。要指定这种编码，可选出符合 $2^k \geqslant m+1$ 的最小 k 值。此类编码的码长是 $m+k+1$，其中“$+1$”表示检测其他校验位的整体奇偶性所用的那一位。然而这样构建出来的编码并不比依照汉明规则构建出来的好，而且有时还更差。
3. 将 k 与 m 视为实数，则下述迭代过程很快就从下方收敛了：

$$k_0 = 0,$$
$$k_{i+1} = \lg\ (k_i + m + 1),\ i = 0,\ 1,\ \ldots$$

   其中 $\lg(x)$ 是以 2 为底 x 的对数。当 $m \geqslant 0$ 时，只需迭代两次，然后求 $\mathrm{ceil}(k_2)$，此值就是正确结果。

   换一种方法。当 $m \geqslant 0$ 时，不难证明：

$$\mathrm{bitsize}(m) \leqslant k \leqslant \mathrm{bitsize}(m) + 1$$

   其中 $\mathrm{bitsize}(m)$ 是将 m 表示为二进制时所需的位元数，例如，$\mathrm{bitsize}(3)=2$，$\mathrm{bitsize}(4)=3$，等等。（这与 5.3.3 节的同名函数不同，那个函数适用于带符号整数。）提示：由 $\mathrm{bitsize}(m) = \lceil \lg(m+1) \rceil = \lfloor \lg(m) + 1 \rfloor$ 可证上式成立，其中 $\lg(0)$ 定为 -1。于是，可先令 $k = \mathrm{bitsize}(m)$，测试这一取值是否正确，如果太小，则为该值加 1。$\mathrm{bitsize}(m)$ 可用前导 0 计数指令算出，此题的计算过程可用代码表示如下：

$$\boldsymbol{k} \leftarrow \boldsymbol{W} - \mathrm{nlz}(\boldsymbol{m})$$

$$\boldsymbol{k} \leftarrow \boldsymbol{k} + (((\mathbf{1} \ll \boldsymbol{k}) - \mathbf{1} - \boldsymbol{k}) \overset{u}{<} \boldsymbol{m})$$

其中 W 为计算机字长，且 $0 \leqslant m \leqslant 2^w - 1$。

4. 证明过程：假设 $d(\boldsymbol{x}, \boldsymbol{z}) > d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, \boldsymbol{z})$，那么必然能找到位置 i，$\boldsymbol{x}$、$\boldsymbol{y}$、$\boldsymbol{z}$ 在该位置的取值，可令 $d(\boldsymbol{x}, \boldsymbol{z})$ 比没有考虑该位置之前多 1，而又能令 $d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, \boldsymbol{z})$ 的值与没有考虑该位置之前相同。于是，前者意味着 $\boldsymbol{x}_i \neq \boldsymbol{z}_i$，而后者却意味着 $\boldsymbol{x}_i = \boldsymbol{y}_i$ 且 $\boldsymbol{y}_i = \boldsymbol{z}_i$，这显然矛盾。所以假设不成立，原命题成立。
5. 给定一套码长为 n 且最小距离为 d 的编码，只需把每个代码字中的“1”与“0”重写为“11”与“00”即可。新编码的码长是 $2n$，最小距离是 $2d$，且代码字的种数和原来相同。
6. 给定一套码长为 n、最小距离为 d 且有 $A(n, d)$ 种代码字的编码，将其按 15.2 节表 15.1 的形式排列出来。随意删掉其中的 $d-1$ 列。新编码的码长为 $n-(d-1)$，且最小距离至少为 1。也就是说，其中每个代码字都互不相同。因而其种数不可能大于 $2^{n-(d-1)}$。因为删除列之后，没能改变代码字的种数，所以原来那套编码至多有 $2^{n-(d-1)}$ 个代码字，于是可证 $A(n,d) \leqslant 2^{n-d+1}$。
7. 根据汉明规则可知，$d=3$ 且包含 2^m 种代码字的编码是完美编码，m 为信息位的个数。将 $A(n, d) = 2^m$ 及 $d=3$ 代入不等式 (6) 右侧，可得

$$2^m \leqslant \frac{2^n}{\binom{n}{0} + \binom{n}{1}} = \frac{2^n}{1+n}$$

将 n 写为 $m+k$，可得

$$2^m \leqslant \frac{2^{m+k}}{1+m+k}$$

将上式两边同除以 2^m，即可变形为不等式 (1)。

8. 这套代码必定包含两个互为反码的位串，所以其代码字的种数为 2。要证明此套编码为完美编码，可设法证明其能达到不等式 (6) 的上界。证明概要如下。n 位二进制整数可理解为从 n 个物体里面选东西，若某位元值为 1，则表示选取对应的物体，若为 0，则表示不选。因此，从 n 个物体中选出 0 至 n 个东西一共有 2^n 种选法，也就是说，$\sum_{i=0}^{n} \binom{n}{i} = 2^n$。若 n 为奇数，则 i 从 0 到 $(n-1)/2$ 的这些项占全部待求和项的一半，由二项式系数的对称性可知 $\binom{n}{i} = \binom{n}{n-i}$，于是，这些项之和也就等于总和的一半。因此 $\sum_{i=0}^{(n-1)/2} \binom{n}{i} = 2^{n-1}$，所以 (6) 式上界是 2。这样就证明了此种编码可以达到 (6) 式的上界。
9. 为便于表述，此题证明过程用到了“等价编码”（equivalence of codes）这一概念。把

一套编码的各列重排（可以拿 15.2 节中的表 15.1 来演示），或将任意一列中的所有位元取反，这样得来的新编码与旧编码显然并无实质区别。若由一套编码经过此种变换而生成另一套，则这两套编码“等价”（equivalent）。由于整套编码是由各代码字所组成的无序集合，所以其书写顺序并不重要。通过将某列的所有位元取反这一方式可把任何一套编码变换为其等价编码，使后者包含一个所有位元均是 0 的代码字。

为行文方便，下列证明过程将演示 $n=9$ 且 $d=6$ 时的情况。

为了不失一般性，可设 0 号代码字（也就是首个代码字，我们称其为 cw_0）是 000 000 000。这样的话，其他代码字必须和 cw_0 有至少 6 个位元不同，也就是说，那些代码字必须至少包含 6 个“1”。

假定此套编码最少有 3 个代码字（下面将证明此论述），那么在非 0 的两个代码字中，1 的个数不可能大于等于 7。因为如果大于等于 7，那么在另一个代码字（其中至少有 6 个 1）中，至少有 4 个位置上的位元值和此代码字重复，且均为 1。于是，两者位元值相等的位置就至少有 4 个，而位元值相异的位置则至多有 5 个（$9-4=5$），这与假设的 $d=6$ 矛盾。因此，除了首个代码字外，其余代码字必须恰好含有 6 个 1。

为了不失一般性，可将各列重排，使前两个代码字变为

cw_0：000 000 000

cw_1：111 111 000

下个代码字 cw_2 在其左侧 6 列中，值为 1 的位元数不可能大于等于 4，因为如果大于等于 4，那么它和 cw_1 就会有 4 个或更多个位置上的位元值相同，而这将导致两者之间位元值不同的位置数小于等于 5。因此，其左侧 6 列中，1 的个数必须小于等于 3。于是，这 6 列中恰好有 3 列是 1。将左边 6 列重排（全部 3 个代码字都要参与重排），使 cw_2 变成

cw_2：111 000 111

与上述推理过程相似，在下一个代码字（cw_3）的左边 3 列和右边 3 列中，1 的总个数不能大于等于 4，因为如果那样的话，就至少有 4 个位置会和 cw_2 的对应位元值重复。因此，在左 3 列和右 3 列中，1 的总个数必须小于等于 3，也就是说，中间 3 列必须是 3 个 1。把这个代码字也和 cw_1 比较一下就能知道，另外 3 个 1 必须在右侧的 3 个位置上。因此 cw_3 就是

cw_3：000 111 111

如果还有下一个代码字的话，那么和 cw_1 相比可知，右侧 3 个位置必须是 3 个 1。而与 cw_2 相比又可知中间 3 个位置也必须是 3 个 1。

10. 因为全 0 和全 1 这两个代码字可以组成一套编码，所以 $A(n, d)$ 显然至少是 2。按照与上题相似的方法来推理，将全 0 的那个代码字记为 cw_0。那么，其他每个代码字中 1 的个数都必须大于 $2n/3$。如果这套编码中有至少 3 个代码字，那么除了 cw_0 之外，还有至少两个代码字，其位元值相同的位置个数必然大于 $2n/3-n/3=n/3$，可参考下图。

$$\underbrace{1111\ldots1111}_{>2n/3}\underbrace{0\ldots0}_{<n/3}$$

（在上图所表示的 cw_1 中，“1”全部排在了左边。现在要把大于 $2n/3$ 个“1”排布在 cw_2 中，而且要尽力避免与 cw_1 中位元值为“1”的那些位置重合。）由于 cw_1 与 cw_2 重合的位置数必然大于 $n/3$，所以两者位元值不同的位置数必然小于 $n-n/3=2n/3$，于是导致整套代码的最小距离小于 $2n/3$。这与题目中所说的 $d>2n/3$ 矛盾，所以编码中至少有3个代码字这一假设不成立。

11. 因为代码字之间最小距离是4，所以此编码是SEC-DED码。为了理解这一点，首先假设两个代码字有一个信息位不同。那么除了这个信息位以外，行奇偶校验位、列奇偶校验位、右下角的校验位也会不同，这将使两者的距离为4。如果两代码字中有两个信息位不同，且均在一行里，那么二者的行奇偶校验位相同，但是会有两列的列奇偶校验位不同。因此其距离也是4。如果两代码字中有两个信息位不同，且均在同一列，那么间距还是4。如果两个值不同的信息位既不同行也不同列，那么这两个代码字的距离就是6。如果有3个信息位不同，那么不难验证，无论其在行列间如何排布，总会有至少一个奇偶校验位不同。因此，间距至少为4。

    若不使用右下角那个位元，则最小距离就是3。此时这套编码不是SEC-DED码，不过仍然是一套SEC码。

    无论右下角那一位检测的是行奇偶校验位之和还是列奇偶校验位之和，其值都等于64个信息位之和除以2的余数，所以在这两种情况下值相同。

    这套编码需要17个校验位，而汉明码则只需8个（参照15.2.3节的表15.3），所以从这个角度看，不是特别高效。

    然而其检测突发错误的效果较好。假设 9×9 矩阵在“位串行通道”（bit serial channel）中按照第0行、第1行、…、第8行的顺序传输。那么，一个位元数小于等于10的序列就可以设法放在同一行或连续两行中，如果是后者，则最多有一个位元重叠。因此，如果传输错误仅发生在这至多10个连续位元中，那么大多数情况下只需检查列奇偶校验位即可侦测错误，如果恰好是首尾两个位元出错，那么检查行奇偶校验位即可。

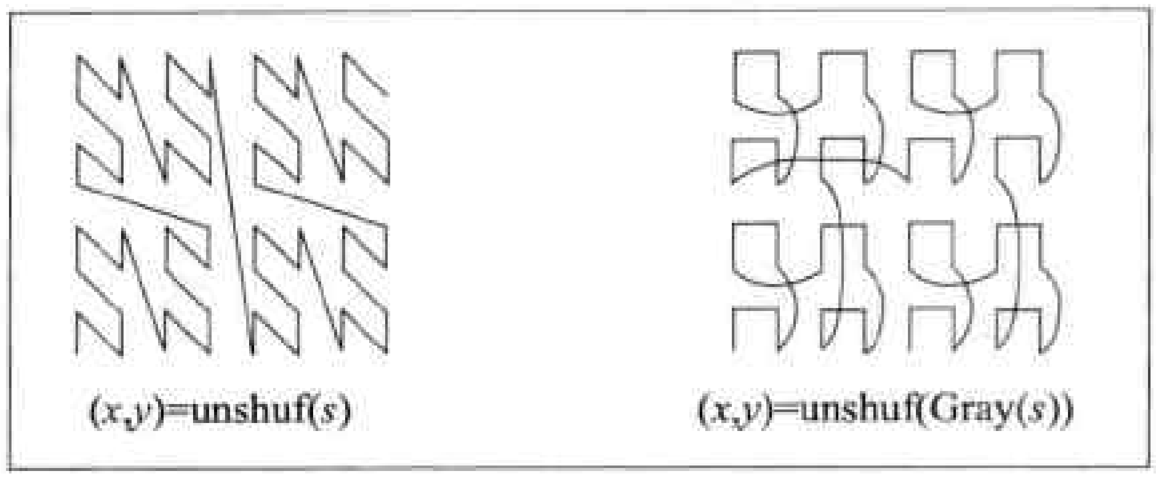
    若有4个位元出错，且恰好排成矩形[⊖]，则此套编码检测不出错误。

第16章

1. 习题2的答案也一并列于此。

    左图所示曲线，在行进过程中的平均跳跃距离约为1.46，而右图的平均跳跃距离约为1.33。因此，至少从曲线上两点的间距这一角度看，使用格雷码似乎能提升“局部性”。

⊖ 意思是：同一行中有两个位元出错，而其上某行或其下某行恰好也在这两个位置上错了。——译者注



在艾兹赫尔·戴克斯特拉（Edsger Dijkstra，1930—2002）的建议下，早期 Algol[1]编译器使用洗牌算法（shuffle algorithm）把矩阵映射到后备存储器（backing store）中。这么做是为了减少求逆矩阵时所执行的分页操作次数。他把这叫做“拉链算法”（zip-fastener algorithm）。似乎许多人都曾独立发现这一算法。

3. 从 s 首位开始，每 3 个位置取 1 个位元，这些位元就构成了 x，从 s 第 2 位开始，每 3 个位置取 1 个位元，就构成了 y，从 s 第 3 位开始，每 3 个位置取 1 个位元，就构成了 z，(x,y,z) 即是三维曲线各点坐标。

第 17 章

1. ±0、±2.0 以及特定的 NaN。
2. 有这样的程序。假设 $x=2^n(1+f)$，那么很容易就能推出：

$$\sqrt{x} = 2^{n/2}(1+f)^{1/2}$$

忽略小数部分之后，我们发现，带偏移量的指数要由原来的 $127+n$ 变成现在的 $127+n/2$。而后者可以写为 $(127+n)/2+127/2$。这样看来，似乎可以把 rep（x）右移一位，并给指数字段加上 63（也就是给 rep（x）加上 0x1F800 000），以此来估算$\sqrt{x}$。下面这个估算公式

$$\mathrm{rep}(\sqrt{x}) \approx k + (\mathrm{rep}(x) \overset{s}{\gg} 1)$$

和正文中提到的平方根倒数估算公式相同，也具备这样一个性质：若能根据 1.0 至 4.0 之间的 x 找到最优常量 k，则此 k 值对所有 x 来说都最优。先找一个能够根据给定 k 值求出最大误差与最小误差的辅助程序，然后拿该程序来试探常量 k 的最优取值，找到最佳值之后，就可以写出下面这段代码了。其中使用了一次“牛顿-拉弗森迭代法”。

```
float asqrt(float x0) {
   union {int ix; float x;};

   x = x0;                        // x can be viewed as int.
   ix = 0x1fbb67a8 + (ix >> 1); // Initial guess.
   x = 0.5f*(x + x0/x);          // Newton step.
   return x;
}
```

[1] Algol 是“算法语言”（ALGOrithmic Language）一词的缩写，该语言适合数值计算。详情参见：http://en.wikipedia.org/wiki/ALGOL。——译者注

对正规数来说，相对误差在0至大约0.000 601之间。在x为无穷或NaN时能算出正确结果（正确结果分别是无穷和NaN）。$x=0$时的结果大约是4.0×10^{-20}。当$x=-0$时，运算结果是1.35×10^{19}，该值非常不合理。当x为正的非规范数时，运算结果要么在前述容错范围内，要么是一个比10^{-19}小的正数。

由于牛顿法那一步用了除法，所以在大多数计算机中，该程序并没有求平方根倒数的那个程序快。

假设再用一次牛顿迭代，那么就能将正规数的相对误差范围缩减为0与0.000 000 23之间，此时的最优常数是0x1FBB 3F80。如果连一次牛顿迭代都不使用，且常数为0x1FBB 4F2E，那么相对误差小于±0.035。这个误差范围几乎和不带牛顿法的平方根倒数程序相同，而且与那个程序一样，也只需要两次整数操作。

3. 有这样的程序，可以按照和上题基本相同的原理，写出正规范数的立方根估值代码。关键语句在于首次估值：

```
i = 0x2a51067f + i/3;        // Initial guess.
```

这样算出来的立方根，其相对误差约为±0.0316。可用下述公式来估算某数除以3的值：

$$\frac{i}{3}\approx\frac{i}{4}+\frac{i}{16}+\frac{i}{64}+\cdots+\frac{i}{65536}$$

（分数中的分母都是2的幂，所以可用右移来实现）。下面这段代码在实现除法时用了7条指令，并略微提升了精确度。（除法技巧请参阅10.18节。）

```
float acbrt(float x0) {
   union {int ix; float x;};

   x = x0;                      // x can be viewed as int.
   ix = ix/4 + ix/16;           // Approximate divide by 3.
   ix = ix + ix/16;
   ix = ix + ix/256;
   ix = 0x2a5137a0 + ix;        // Initial guess.
   x = 0.33333333f*(2.0f*x + x0/(x*x));  // Newton step.
   return x;
}
```

尽管不用除以3（改用7条基本整数指令实现），但是牛顿法那一步依然要执行除法及另外4条指令。相对误差范围是0至大约+0.001 03。这样看来，此方法并没有平方根倒数与平方根估算法那么理想，不过在某些情况下依然可以使用。

如果还用这个常数，并且再执行一次牛顿迭代步骤，那么相对误差范围就是0至大约+0.000 001 16。

4. 有这样的程序。下面列出的这个程序可以求双精度浮点数的平方根倒数，其精确度约为±3.5%[⊖]。只需再用一两次牛顿-拉弗森迭代，即可提升精度。用0x5fe80...0作常

⊖ 也就是说，计算值与真实值之差占真实值的百分比约为−3.5～+3.5之间。——译者注

量，运算结果的相对误差范围是 0 至大约 +0.887，如果使用 0x5fe618fdf80...0 作为常量，那么相对误差范围是 0 至大约 −0.0613。

```
double rsqrtd(double x0) {
   union {long long ix; double x;};

   x = x0;
   ix = 0x5fe6ec85e8000000LL - (ix >> 1);
   return x;
}
```

第 18 章

1. 设 $f(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a^0$。像这样的多项式，当 x 趋近于无穷时，$f(x)$ 在量值上也单调趋近于无穷。（如果 x 足够大，那么第一项在量值上就超过其他所有项之和了。）

   设 x_0 为一个整数，当 $x\geqslant x_0$ 时，$|f(x)|\geqslant 2$ 这一不等式对所有 x 都成立。设 $f(x_0)=k$，设 r 为任意正整数。则 $|k|\geqslant 2$，且

   $$\begin{aligned}|f(x_0+rk)| &= |a_n(x_0+rk)^n+a_{n-1}(x_0+rk)^{n-1}+\cdots+a_0| \\ &= |f(x_0)+rk\ \text{的倍数}| \\ &= |k+rk\ \text{的倍数}|\end{aligned}$$

   因此，随着 r 增大，由 $|f(x_0+rk)|$ 所表示的合数也会越来越大。因此，在 $|f(x)|$ 的取值中，将会有无穷多个合数。

   此定理还有一种叙述方式：即便参数取得再大，也找不到其值仅为素数的“单变量非常数多项式”(non-constant polynomial in one variable)。

   例如，设 $f(x)=x^2+x+41$。那么 $f(1)=43$ 且

   $$\begin{aligned}f(1+43r) &= (1+43r)^2+(1+43r)+41 \\ &= (1+86r+43^2r^2)+(1+43r)+41 \\ &= 1+1+41+86r+43^2r^2+43r \\ &= 43+(2+43r+1)\cdot 43r\end{aligned}$$

   显然，函数值总是 43 的倍数，而且随着 r 增大，函数值也越来越大。

2. 假设 p 是合数，那么同余式可写成

   $$(p-1)!=pk-1$$

   其中 k 为某个整数。设 a 是 p 的“真因子”(proper factor)[⊖]。那么 a 就必然能整除等式左方的数，但是 a 却不能整除等式右边的数，所以上述等式显然不成立，故原命题成立。

   当 p 为 1、2、3 时，显然可知该定理成立。假设 p 为大于 3 的素数，那么阶乘可展

⊖ 除去该整数本身，其余因子均可称为“真因子”。在本例中，a 是 p 的因子，但不能与 p 相等。——译者注

开为：

$$(p-1)!=(p-1)(p-2)\cdots(3)(2)$$

第一项是 $p-1$，该值与 -1 关于模 p 同余。其他各项均与 p 互质，因而那些项对于模 p 来说都有“乘法逆元素”（参见 10.16 节定理 MI），每一项的乘法逆元素各不相同，而且都不等于该项本身。

为了证明阶乘展开式中的每一项对于模素数 p 的乘法逆元素都不能与其自身相等（除了 1 和 $p-1$ 这两项外），我们假设 $a^2\equiv 1 \pmod p$。那么 $a^2-1\equiv 0 \pmod p$，于是 $(a-1)(a+1)\equiv 0 \pmod p$。由于 p 是素数，所以 $a-1$ 或 $a+1$ 必须能为 p 所整除。如果是前者，那么就可推出 $a\equiv 1 \pmod p$，如果是后者，那么就可推出 $a\equiv -1\equiv p-1 \pmod p$。而这两者均不成立[㊀]，于是刚才的假设不成立，所以说，阶乘展开式里的中间各项对于模素数 p 的乘法逆元素都不可能是其自身。

因此，可以把 $p-2$，$p-3$，…，2 等整数两两配对，令每一对的乘积都与 1 对模 p 同余。也就是说：

$$(p-1)!=(p-1)(ab)(cd)\cdots$$

其中 a 与 b 是乘法逆元素，c 与 d 是乘法逆元素，等等。这样一来，就可得到

$$(p-1)!\equiv(-1)(1)(1)\equiv -1 \pmod p$$

例如，当 $p=11$ 时，$10! \pmod{11}\equiv 10\cdot 9\cdot 8\cdot 7\cdot 6\cdot 5\cdot 4\cdot 3\cdot 2 \pmod{11}\equiv 10\cdot(9\cdot 5)(8\cdot 7)(6\cdot 2)(4\cdot 3) \pmod{11}\equiv(-1)(1)(1)(1)(1) \pmod{11}\equiv -1 \pmod{11}$。

该定理以约翰·威尔逊（John Wilson，英格兰数学家，1741—1793）命名，威尔逊是英格兰数学家爱德华·华林（Edward Waring，约 1736—1798）的学生。华林于 1770 年描述了该定理，但未加证明。拉格朗日（Joseph-Louis Lagrange，法国数学家，1736—1813）于 1773 年最先发表了证明。此定理在公元 1000 年左右就为中世纪欧洲人所知[㊁]。

3. 若 $n=ab$，且 a 与 b 不相等，同时两数又都不等于 1 或 n，那么显然可知 a 与 b 都比 n 小，于是两数必然包含在 $(n-1)!$ 的阶乘展开式中。因此，n 可以整除 $(n-1)!$。

   若 $n=a^2$ 且 $a>2$，那么 $a^2=n>2a$，由此可知 a 与 $2a$ 均在 $(n-1)!$ 的阶乘展开式中，因此，a^2 可以整除 $(n-1)!$。

4. 有这样一类问题，其求解过程虽然是非正式证明，然而我们在计算时所领略到的数学之美比正式证明更甚，本题可能就属于这种情况。

   根据米尔斯定理，有实数 θ 可令 $\lfloor \theta^{3^n} \rfloor$ 在 $n\geqslant 1$ 时恒为素数。首先试试能不能找到一个 θ

㊀ 由命题可知，a 在 2 至 $p-2$ 之间，如果 $a\equiv 1 \pmod p$ 成立，那就意味着 a 除以 p 余 1，而在 2 至 $p-2$ 这个取值范围里找不到符合此条件的 a 值；如果 $a\equiv -1\equiv p-1 \pmod p$ 成立，那就意味着 a 除以 p 余 $p-1$，同样也不可能在 2 至 $p-2$ 之间找到这样的 a 值。所以两个推论都不成立。——译者注

㊁ 阿拉伯数学家海什木（Alhazen，965—约 1040）在求解同余问题时，曾用到了后世称为威尔逊定理的算式。详情参见：http：//en.wikipedia.org/wiki/Ibn_al-Haytham#Number_theory。——译者注

值，令 $n=1$ 时根据定理所求出的素数为 2。如果能，那么

$$\lfloor \theta^{3^1} \rfloor = 2$$

于是

$$2 \leqslant \theta^{3^1} < 3\text{，或者说} \tag{1}$$

$$2^{1/3} \leqslant \theta < 3^{1/3}\text{，也就是说}$$

$$1.2599\ldots \leqslant \theta < 1.4422\ldots$$

将不等式（1）取立方，可得

$$8 \leqslant \theta^{3^2} < 27 \tag{2}$$

在这个范围内能找到至少一个素数。（按照我们的假设，2^3 与 $(2+1)^3$ 之间总有一个素数。）接下来再用 11 这个素数试一试。如果有这样的 θ 值，那么把不等式（2）继续缩窄为下式，即可得出 $\lfloor \theta^{3^2} \rfloor = 11$：

$$11 \leqslant \theta^{3^2} < 12 \tag{3}$$

现在给（3）式取立方，得出

$$1331 \leqslant \theta^{3^3} < 1728 \tag{4}$$

我们能够确定 1331 至 1728 之间必然有素数。假设选定其中最小的素数，也就是 1361。那么，继续缩窄（4）式，可得

$$1361 \leqslant \theta^{3^3} < 1362$$

至此已经验证，当 n 为 1、2、3 时，能找到实数 θ，使得 $\lfloor \theta^{3^n} \rfloor$ 为素数，将 1361 与 1362 开 27 次方根，可知 $n=3$ 时的 θ 位于 1.306 37 与 1.306 42 之间。

此过程显然可以一直持续下去。最后可以证明 θ 趋近于某个极限值，不过这一论述并非绝对必要。其实只要能证明在 n 趋于极限时，θ 是某个有限范围内的任意数，那么就可以证明米尔斯定理了。

通过上述计算过程可以看出，米尔斯定理略显穿凿。若想令该定理成为素数公式，就必须根据素数来决定 θ 才行。这与 18.1 节根据常数 a 来求素数的公式类似：

$$a = 0.203\,005\,000\,700\,011\,000\,013\ldots$$

该定理显然与素数没有多大关系。只要给定一个足够稠密的递增序列，就会有类似定理出现。

上述步骤可以算出符合米尔斯定理的最小 θ 值。该值有时称为米尔斯常数（Mills' constant），现在已经计算到超过小数点后 6850 位了［CC］。

5. 假设有符合下式的 a、b、c、d 4 个整数：

$$a + b\sqrt{-5}(c + b\sqrt{-5}) = 2 \tag{5}$$

那么将等式左侧展开，令其中的实部与虚部分别与右侧相等：

$$ac - 5bd = 2 \tag{6}$$

$$ad + bc = 0 \tag{7}$$

如果 c 等于 0，那么根据（6）式可得 $-5bd=2$，该式无整数解，所以，c 显然不为 0。

b 也不能等于 0，因为如果 $b=0$，那么根据（7）式，a 和 d 至少有一个为 0。然而 $a=0$ 又无法满足等式（5），所以只可能是 $d=0$。这样一来，（5）式就变为 $ac=2$，于是，（5）式中的一个因子就必然为“单位元素”，这与题目所要求的因子分解式不符。

由（7）式可知，$abd+b^2c=0$。由（6）式可知，$a^2c-5abd=2a$。结合[1]上述两式，可得 $a^2c+5b^2c=2a$，或者说：

$$a^2+5b^2=2a/c \tag{8}$$

（刚才已经证明了 c 不为 0）。（8）式左侧至少是 a^2+5，而 a 和 c 无论取什么值，等式左侧的数都要比等式右侧的 $2a/c$ 大[2]。

为了证明 3 是素数，也可以推出与（8）式类似的式子来，其中 $b \neq 0$ 且 $c \neq 0$：

$$a^2+5b^2=3a/c$$

把 6 这个数分解为质因子，有两种写法：

$$6=2\cdot 3=(1+\sqrt{-5})(1-\sqrt{-5})$$

我们没有证明 $1\pm\sqrt{-5}$ 为何是“素数”。其论证过程与证明 2 和 3 是素数时相似（只是有些长），其实，要想证明 6 在这个“环”中的质因子分解式并不唯一，无需论证 $1\pm\sqrt{-5}$ 是素数。因为不管 $1\pm\sqrt{-5}$ 是不是素数，这些数最后总能分解为素数，而其质因子分解式绝不可能是 $2\cdot 3$。

[1] 将两式相加，其左侧为 $abd+b^2c+a^2c-5abd=a^2c+b^2c-4abd$。而根据（7）式可知 $ad=-bc$，所以 $-4abd=(ad)\cdot(-4b)=(-bc)\cdot(-4b)=4b^2c$，于是 $a^2c+b^2c-4abd=a^2c+b^2c+4b^2c=a^2c+5b^2c$；而两式相加的右侧是 $0+2a=2a$。综上可推出 $a^2c+5b^2c=2a$。——译者注

[2] 因为 $a^2+5>|2a|$ 必然成立，而如果 a、c 一正一负，那么 $2a/c$ 就是负数，而 a^2+5 是正数，显然大于 $2a/c$；如果 a、c 正负号相同，那么 $|2a|>=2a/c$，于是 $a^2+5>2a/c$；如果 a 是 0，那么 $0^2+5>2\cdot 0/c$。所以 a^2+5 总大于 $2a/c$。——译者注

附录 A
4 位计算机算术运算表

在附录 A 的表格中，下划线表示带符号溢出（signed overflow）。例如，在表 A.1 中，7+1=8，而 4 位计算机无法用带符号数来表示 8 这个值，所以就会出现带符号溢出。

表 A.1　加法表

										−8	−7	−6	−5	−4	−3	−2	−1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	1	1	2	3	4	5	6	7	$\underline{8}$	9	A	B	C	D	E	F	10
	2	2	3	4	5	6	7	$\underline{8}$	$\underline{9}$	A	B	C	D	E	F	10	11
	3	3	4	5	6	7	$\underline{8}$	$\underline{9}$	$\underline{A}$	B	C	D	E	F	10	11	12
	4	4	5	6	7	$\underline{8}$	$\underline{9}$	$\underline{A}$	$\underline{B}$	C	D	E	F	10	11	12	13
	5	5	6	7	$\underline{8}$	$\underline{9}$	$\underline{A}$	$\underline{B}$	$\underline{C}$	D	E	F	10	11	12	13	14
	6	6	7	$\underline{8}$	$\underline{9}$	$\underline{A}$	$\underline{B}$	$\underline{C}$	$\underline{D}$	E	F	10	11	12	13	14	15
	7	7	$\underline{8}$	$\underline{9}$	$\underline{A}$	$\underline{B}$	$\underline{C}$	$\underline{D}$	$\underline{E}$	F	10	11	12	13	14	15	16
−8	8	8	9	A	B	C	D	E	F	$\underline{10}$	$\underline{11}$	$\underline{12}$	$\underline{13}$	$\underline{14}$	$\underline{15}$	$\underline{16}$	$\underline{17}$
−7	9	9	A	B	C	D	E	F	10	$\underline{11}$	$\underline{12}$	$\underline{13}$	$\underline{14}$	$\underline{15}$	$\underline{16}$	$\underline{17}$	18
−6	A	A	B	C	D	E	F	10	11	$\underline{12}$	$\underline{13}$	$\underline{14}$	$\underline{15}$	$\underline{16}$	$\underline{17}$	18	19
−5	B	B	C	D	E	F	10	11	12	$\underline{13}$	$\underline{14}$	$\underline{15}$	$\underline{16}$	$\underline{17}$	18	19	1A
−4	C	C	D	E	F	10	11	12	13	$\underline{14}$	$\underline{15}$	$\underline{16}$	$\underline{17}$	18	19	1A	1B
−3	D	D	E	F	10	11	12	13	14	$\underline{15}$	$\underline{16}$	$\underline{17}$	18	19	1A	1B	1C
−2	E	E	F	10	11	12	13	14	15	$\underline{16}$	$\underline{17}$	18	19	1A	1B	1C	1D
−1	F	F	10	11	12	13	14	15	16	$\underline{17}$	18	19	1A	1B	1C	1D	1E

在 A.2 这张减法表中，我们假定 $a \cdot b$ 的进位与 $a+\overline{b}+1$ 相同，也就是说，减法中的“进位”，意思是“不借位”。

对于乘法（表 A.3 与表 A.4）来说，溢出意味着结果无法表示成 4 位二进制数。在带符号乘法中（如表 A.3 所示），这也就等于说 8 位计算结果的前 5 位不全是 1 或不全是 0。

表 A.2 减法表（行减列）

										−8	−7	−6	−5	−4	−3	−2	−1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1
	1	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2
	2	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3
	3	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4
	4	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5
	5	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6
	6	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7
	7	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8
−8	8	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9
−7	9	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A
−6	A	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B
−5	B	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C
−4	C	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D
−3	D	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E
−2	E	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F
−1	F	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10

表 A.3 带符号乘法表

										−8	−7	−6	−5	−4	−3	−2	−1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	2	3	4	5	6	7	F8	F9	FA	FB	FC	FD	FE	FF
	2	0	2	4	6	8	A	C	E	F0	F2	F4	F6	F8	FA	FC	FE
	3	0	3	6	9	C	F	12	15	E8	EB	EE	F1	F4	F7	FA	FD
	4	0	4	8	C	10	14	18	1C	E0	E4	E8	EC	F0	F4	F8	FC
	5	0	5	A	F	14	19	1E	23	D8	DD	E2	E7	EC	F1	F6	FB
	6	0	6	C	12	18	1E	24	2A	D0	D6	DC	E2	E8	EE	F4	FA
	7	0	7	E	15	1C	23	2A	31	C8	CF	D6	DD	E4	EB	F2	F9
−8	8	0	F8	F0	E8	E0	D8	D0	C8	40	38	30	28	20	18	10	8
−7	9	0	F9	F2	EB	E4	DD	D6	CF	38	31	2A	23	1C	15	E	7
−6	A	0	FA	F4	EE	E8	E2	DC	D6	30	2A	24	1E	19	12	C	6
−5	B	0	FB	F6	F1	EC	E7	E2	DD	28	23	1E	19	14	F	A	5
−4	C	0	FC	F8	F4	F0	EC	E8	E4	20	1C	18	14	10	C	8	4
−3	D	0	FD	FA	F7	F4	F1	EE	EB	18	15	12	F	C	9	6	3
−2	E	0	FE	FC	FA	F8	F6	F4	F2	10	E	C	A	8	6	4	2
−1	F	0	FF	FE	FD	FC	FB	FA	F9	8	7	6	5	4	3	2	1

表 A.4 无符号乘法表

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

表 A.5 及表 A.6 用于计算常见的“向 0 取整式除法”(truncating division，又名“截取除法”)。表 A.5 将绝对值最大的负数与−1 相除之商记为 8，并认定此时会产生溢出，然而在这种情况下，大部分计算机要么将运算结果视为“未定义”(undefined)，要么禁用该操作。

表 A.5 带符号短除法表(行÷列)

										−8	−7	−6	−5	−4	−3	−2	−1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	–	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	–	1	0	0	0	0	0	0	0	0	0	0	0	0	0	F
	2	–	2	1	0	0	0	0	0	0	0	0	0	0	0	F	E
	3	–	3	1	1	0	0	0	0	0	0	0	0	0	F	F	D
	4	–	4	2	1	1	0	0	0	0	0	0	0	F	F	E	C
	5	–	5	2	1	1	1	0	0	0	0	0	F	F	F	E	B
	6	–	6	3	2	1	1	1	0	0	0	F	F	F	F	D	A
	7	–	7	3	2	1	1	1	1	0	F	F	F	F	E	D	9
−8	8	–	8	C	E	E	F	F	F	1	1	1	1	2	2	4	8
−7	9	–	9	D	E	F	F	F	0	1	1	1	1	1	2	3	7
−6	A	–	A	D	E	F	F	F	0	0	0	1	1	1	2	3	6
−5	B	–	B	E	F	F	F	0	0	0	0	0	1	1	1	2	5
−4	C	–	C	E	F	F	0	0	0	0	0	0	0	1	1	2	4
−3	D	–	D	F	F	0	0	0	0	0	0	0	0	0	1	1	3
−2	E	–	E	F	0	0	0	0	0	0	0	0	0	0	0	1	2
−1	F	–	F	0	0	0	0	0	0	0	0	0	0	0	0	0	1

表 A.6 无符号短除法（行 ÷ 列）

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	–	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	–	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	–	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	–	3	1	1	0	0	0	0	0	0	0	0	0	0	0	0
4	–	4	2	1	1	0	0	0	0	0	0	0	0	0	0	0
5	–	5	2	1	1	1	0	0	0	0	0	0	0	0	0	0
6	–	6	3	2	1	1	1	0	0	0	0	0	0	0	0	0
7	–	7	3	2	1	1	1	1	0	0	0	0	0	0	0	0
8	–	8	4	2	2	1	1	1	1	0	0	0	0	0	0	0
9	–	9	4	3	2	1	1	1	1	1	0	0	0	0	0	0
A	–	A	5	3	2	2	1	1	1	1	1	0	0	0	0	0
B	–	B	5	3	2	2	1	1	1	1	1	1	0	0	0	0
C	–	C	6	4	3	2	2	1	1	1	1	1	1	0	0	0
D	–	D	6	4	3	2	2	1	1	1	1	1	1	1	0	0
E	–	E	7	4	3	2	2	2	1	1	1	1	1	1	1	0
F	–	F	7	5	3	3	2	2	1	1	1	1	1	1	1	1

表 A.7 与 A.8 列出了常见的“向 0 取整式除法”之余数。表 A.7 将绝对值最大的负数除以 −1 之后的余数记为 0，但实际上，大多数计算机在这种情况下，要么将结果视为“未定义”，要么禁用该操作。

表 A.7 带符号短除法余数表（行 ÷ 列）

									−8	−7	−6	−5	−4	−3	−2	−1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	–	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	–	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
	2	–	0	0	2	2	2	2	2	2	2	2	2	2	2	0	0
	3	–	0	1	0	3	3	3	3	3	3	3	3	3	0	1	0
	4	–	0	0	1	0	4	4	4	4	4	4	4	0	1	0	0
	5	–	0	1	2	1	0	5	5	5	5	5	0	1	2	1	0
	6	–	0	0	0	2	1	0	6	6	6	0	1	2	0	0	0
	7	–	0	1	1	3	2	1	0	7	0	1	2	3	1	1	0
−8	8	–	0	0	E	0	D	E	F	0	F	E	D	0	E	0	0
−7	9	–	0	F	F	D	E	F	0	9	0	F	E	D	F	F	0
−6	A	–	0	0	0	E	F	0	A	A	A	0	F	E	0	0	0
−5	B	–	0	F	E	F	0	B	B	B	B	B	0	F	E	F	0
−4	C	–	0	0	F	0	C	C	C	C	C	C	C	0	F	0	0
−3	D	–	0	F	0	D	D	D	D	D	D	D	D	D	0	F	0
−2	E	–	0	0	E	E	E	E	E	E	E	E	E	E	E	0	0
−1	F	–	0	F	F	F	F	F	F	F	F	F	F	F	F	F	0

表 A.8 无符号短除法余数表（行÷列）

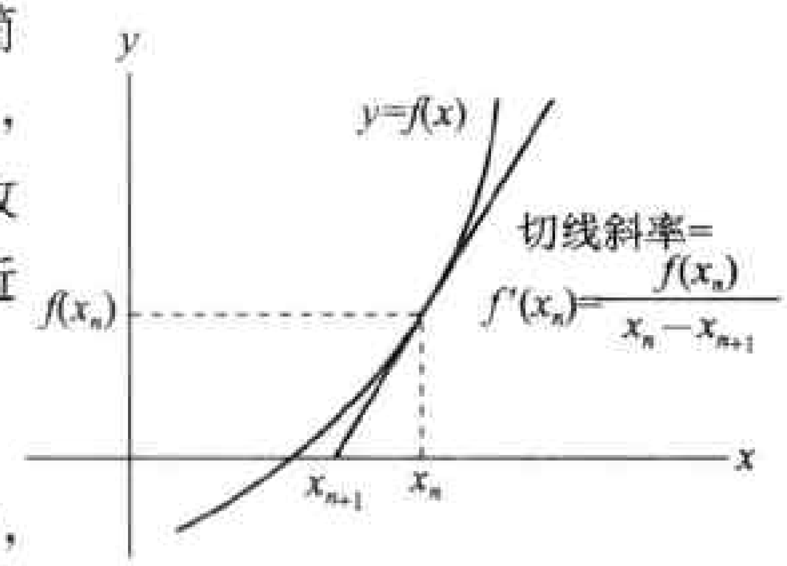
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	–	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	–	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	–	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2
3	–	0	1	0	3	3	3	3	3	3	3	3	3	3	3	3
4	–	0	0	1	0	4	4	4	4	4	4	4	4	4	4	4
5	–	0	1	2	1	0	5	5	5	5	5	5	5	5	5	5
6	–	0	0	0	2	1	0	6	6	6	6	6	6	6	6	6
7	–	0	1	1	3	2	1	0	7	7	7	7	7	7	7	7
8	–	0	0	2	0	3	2	1	0	8	8	8	8	8	8	8
9	–	0	1	0	1	4	3	2	1	0	9	9	9	9	9	9
A	–	0	0	1	2	0	4	3	2	1	0	A	A	A	A	A
B	–	0	1	2	3	1	5	4	3	2	1	0	B	B	B	B
C	–	0	0	0	0	2	0	5	4	3	2	1	0	C	C	C
D	–	0	1	1	1	3	1	6	5	4	3	2	1	0	D	D
E	–	0	0	2	2	4	2	0	6	5	4	3	2	1	0	E
F	–	0	1	0	3	0	3	1	7	6	5	4	3	2	1	0

附录 B
牛顿法

此处简要回顾牛顿法：假定有一个可微分的函数 f，此函数的变量 x 为实数，现在要算出等式 $f(x)=0$ 中的 x 值。只要估算出与 f 的真实根值相近的值 x_n，那么将其代入牛顿法公式，即可在适当条件下求出下一个更为接近的估算值 x_{n+1}：

$$x_{n+1}=x_n \frac{f\ (x_n)}{f'\ (x_n)}$$

式中 $f'(x_n)$ 是函数 f 在 $x=x_n$处的导数[⊖]。参看下图即可理解此公式的推导原理（从 $f'(x_n)=f(x_n)/(x_n-x_{n+1})$这一等式中解出 x_{n+1}）。



只要首次估值离真实值相当接近，那么对多项式等简单的“良态函数”（well-behaved function）运用牛顿法，效果会很好。一旦估值足够接近，牛顿法就会呈平方级收敛。也就是说，若 r 为根的准确值，而 x_n 为与之足够接近的估算值，那么就有

$$|x_{n+1}-r|\leqslant(x_n-r)^2$$

因此，每轮迭代之后，精确数位的个数会翻倍。（例如，假设 $|x_n-r|\leqslant 0.001$，那么 $|x_{n+1}-r|\leqslant 0.000\,001$）。

如果第一次估算值离真实值太远，那么迭代过程也许收敛得很慢，可能会发散到无穷远，可能会收敛至一个不是最接近于首次估值的根，还可能在某些值之间无限循环。

这样的论述很模糊，因为其中用了诸如“适当的条件”、“良态”、“足够接近”等措辞。至于更为精确的论述，还请参考大学一年级用的微积分教科书，几乎每一本里面都能找到。

尽管使用牛顿法存在上述问题，但在整数范围内还是可以偶尔一用的。若想判明牛顿法能否适用于某个特定函数，还得像 11.1 节求整数平方根那样亲自尝试才行。

表 B.1 列出了几个由牛顿法推出的迭代公式，这些公式可以用来求解特定的数值。

⊖ 导数就是下图中的切线斜率。——译者注

第一栏表示待计算的数，第二栏列出了以第一栏中的数为其一个根㊀的函数，第三栏列出了该函数所对应的牛顿迭代公式中等号右边的部分㊁。

顺便说一句，并不是总能找到可以表示待求数值的好函数。当然，有很多函数的根中都包含待求的那个数，然而其中只有为数不多的几个能够推出有用的迭代公式。一般情况下，函数所描述的运算方式应该与待求数值中的运算互逆。例如，要求出$\sqrt{a}$的值，就要使用 $f(x)=x^2-a$ 这个函数；想求出 $\log_2 a$ 的值，那就使用 $f(x)=2^x-a$ 这个函数，依此类推㊂。

表 B.1 用牛顿法计算特定数值时所用的函数与迭代公式

待计算的量	函数	迭代公式
$\sqrt{a}$	x^2-a	$\frac{1}{2}\left(x_n+\frac{a}{x_n}\right)$
$\sqrt[3]{a}$	x^3-a	$\frac{1}{3}\left(2x_n+\frac{a}{x_n^2}\right)$
$\frac{1}{\sqrt{a}}$	$x^{-2}-a$	$\frac{x_n}{2}(3-ax_n^2)$
$\frac{1}{a}$	$x^{-1}-a$	$x_n(2-ax_n)$
$\log_2 a$	2^x-a	$x_n+\frac{1}{\ln 2}\left(\frac{a}{2^{x_n}}-1\right)$

在计算 $\log_2 a$ 时，即便把迭代公式中第二项的乘数由 1/ln2 改为其他值（如 1，2），该公式也能收敛，但是会收敛得更慢些。在某些场合，可以使用 3/2 或 23/16 这样的近似值（$1/\ln 2\approx 1.4427$）。

㊀ 第二栏中的函数，可能有多个根，例如 $x^2-a=0$，在 a 大于 0 的情况下有两个实数根，一个是$\sqrt{a}$，另一个是$-\sqrt{a}$，而第一栏所列的$\sqrt{a}$只是二者之一。——译者注

㊁ 等号左边总是 x_{n+1}。——译者注

㊂ 作为牛顿法特例的那个求平方根函数，巴比伦人在四千多年前就已经知道了。

附录 C

各种离散函数图像

本附录描绘了若干离散函数图像，这些图均由 Mathematica 生成。每个函数均给出两张图像，一张描述字长为 3 位时的情况，另一张描述字长为 5 位时的情况。笔者根据 Guy Steele 先生的提议，增补了此附录。

C.1 参数为整数的逻辑操作

本节以三维图像描绘了参数 x、y 均为整数的 and (x,y)、or (x,y)、xor (x,y) 函数。这 3 个函数的图像分别参见图 C.1、图 C.2 和图 C.3。

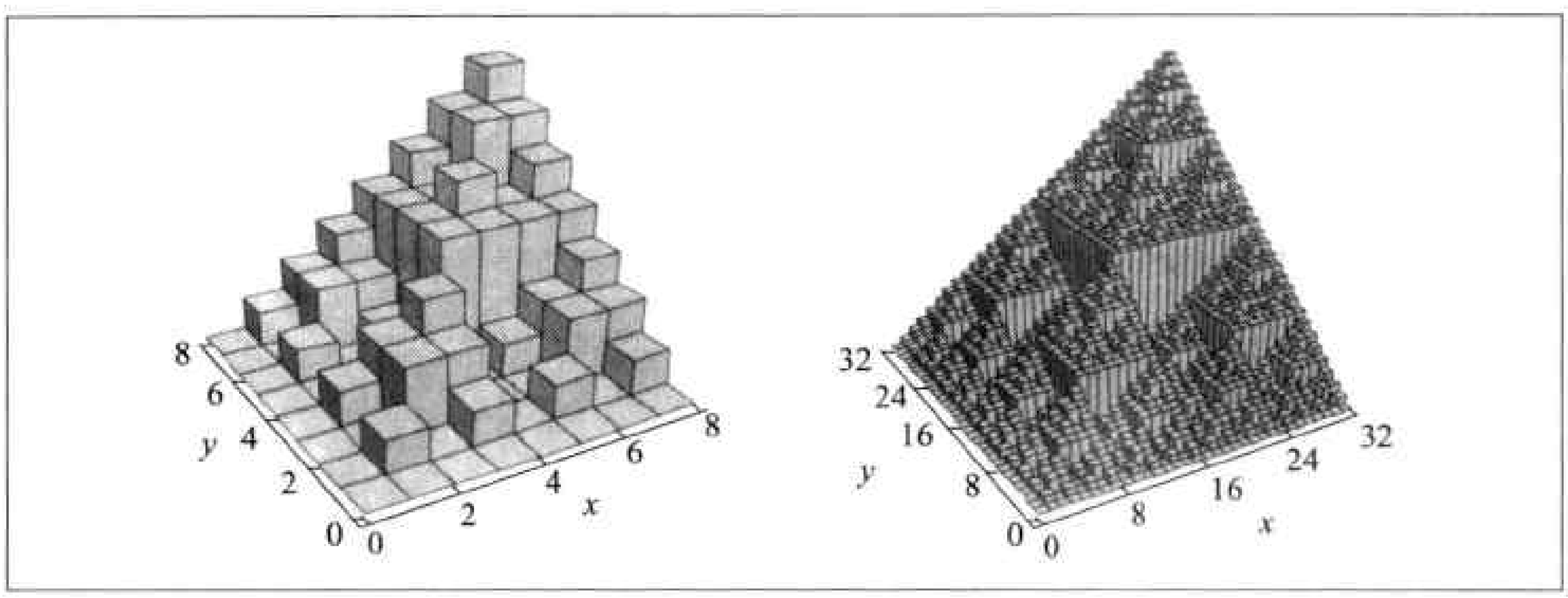


图 C.1 “逻辑与”函数的图像

在图 C.3 中，几乎有一半函数值都隐藏在由 $x=\overline{y}$ 所确定的“对角面”下方。

在 and (x,y) 函数的图像中，显然能看出某种“自我相似”（self-similar）的三角形排列图样，也可以把这叫做“分形”（fractal）[⊖]。假如以平行于 y 轴的视线从正面观察

⊖ “自我相似”也叫“自相似”，如果某物体和其自身的一部分完全或几乎相似，则该物体就是自相似的。它是“分形”的重要特征。这两个概念可分别参见：http://zh.wikipedia.org/wiki/自相似、http://zh.wikipedia.org/wiki/分形。——译者注

and (x,y) 函数，并且把变量的取值上限设为更大的整数，那么函数图像就会如图 C.4 所示。

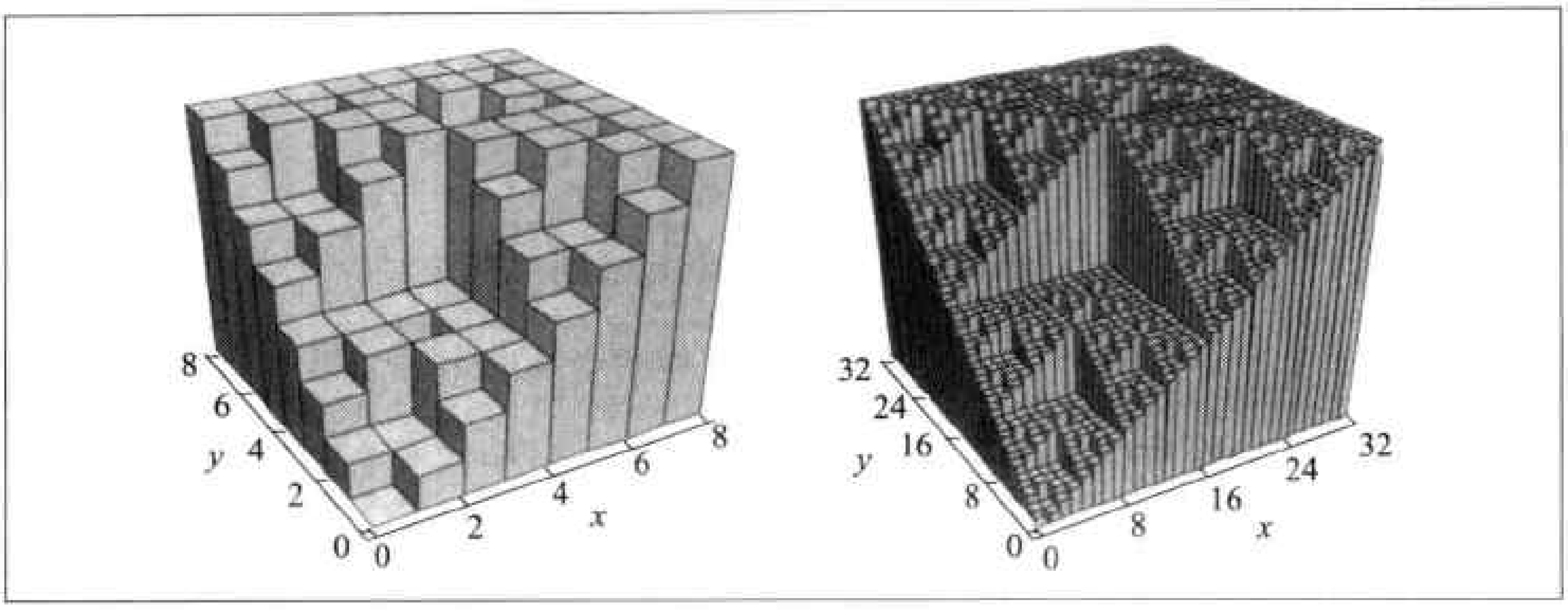


图 C.2 “逻辑或”函数的图像

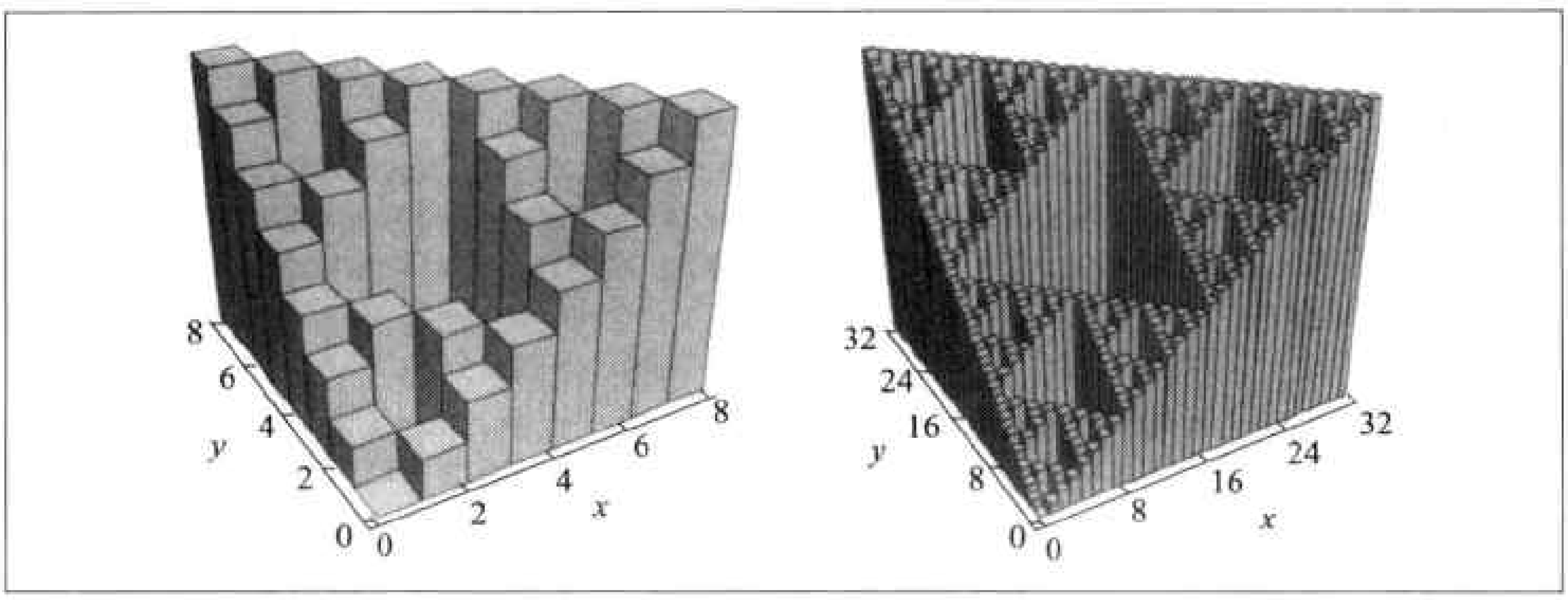


图 C.3 “逻辑异或”函数的图像

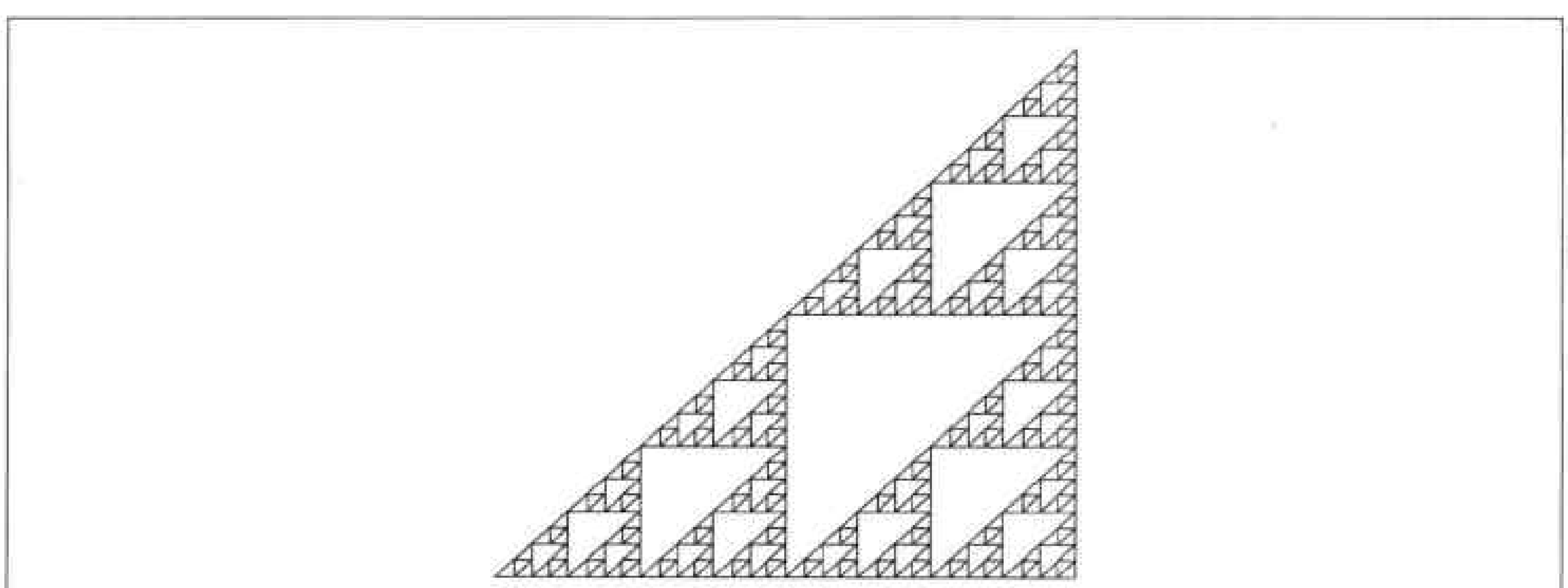
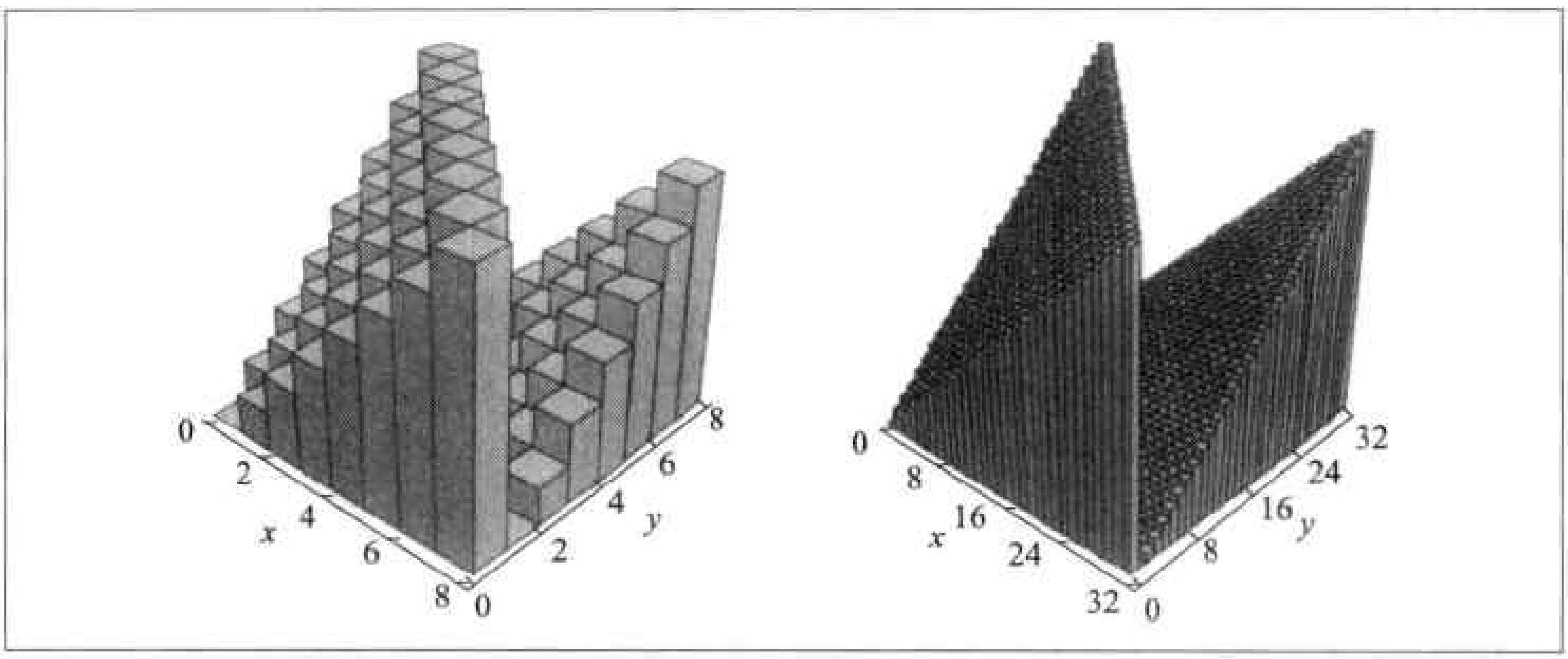


图 C.4 由 and (x,y) 函数所生成的自我相似图样

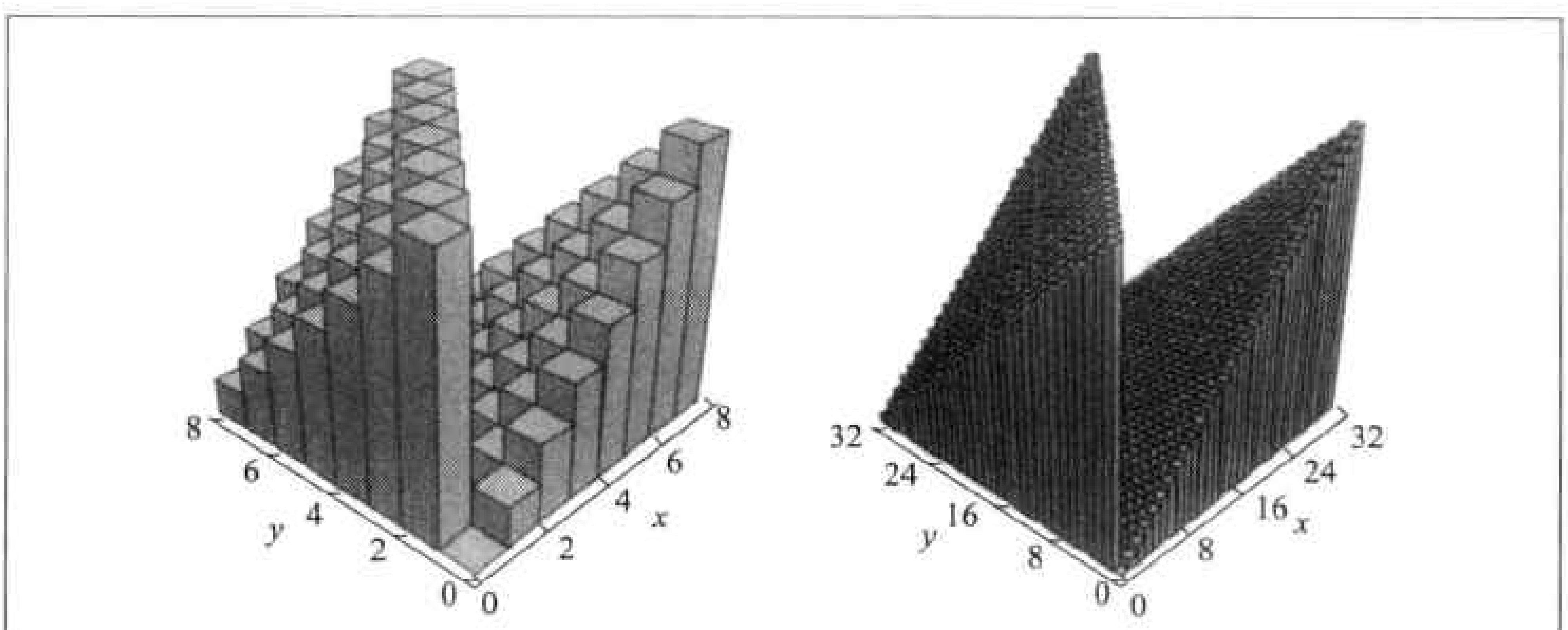
这很像谢尔宾斯基[⊖]三角形（Sierpinski triangle）[Sagan]，区别在于图C.4用的是直角三角形，而谢尔宾斯基三角形是等边三角形。在图C.3中，如果极力扩大变量的取值范围，那么函数图像中的斜面显然会呈现出谢尔宾斯基三角形图样。

C.2 加法、减法、乘法函数的图像

本节的图C.5至图C.9分别描绘了加法、减法以及三种乘法函数的图像，其变量均为无符号整数，按“计算机算术”规则运算。请注意，在加法函数的图像中，原点位于左远角（far-left corner）。



图C.5 **x+y** 的图像（按计算机算术规则运算）



图C.6 **x−y** 的图像（按计算机算术规则运算）

⊖ 瓦茨瓦夫·谢尔宾斯基（1882—1969），波兰数学家，对集合论、数论、函数理论、拓扑论贡献颇多。详情参见：https://zh.wikipedia.org/wiki/瓦茨瓦夫·谢尔宾斯基。——译者注

图 C.7 压缩了垂直方向上的刻度，在左图中，最高峰的高度是 7 · 7=49。

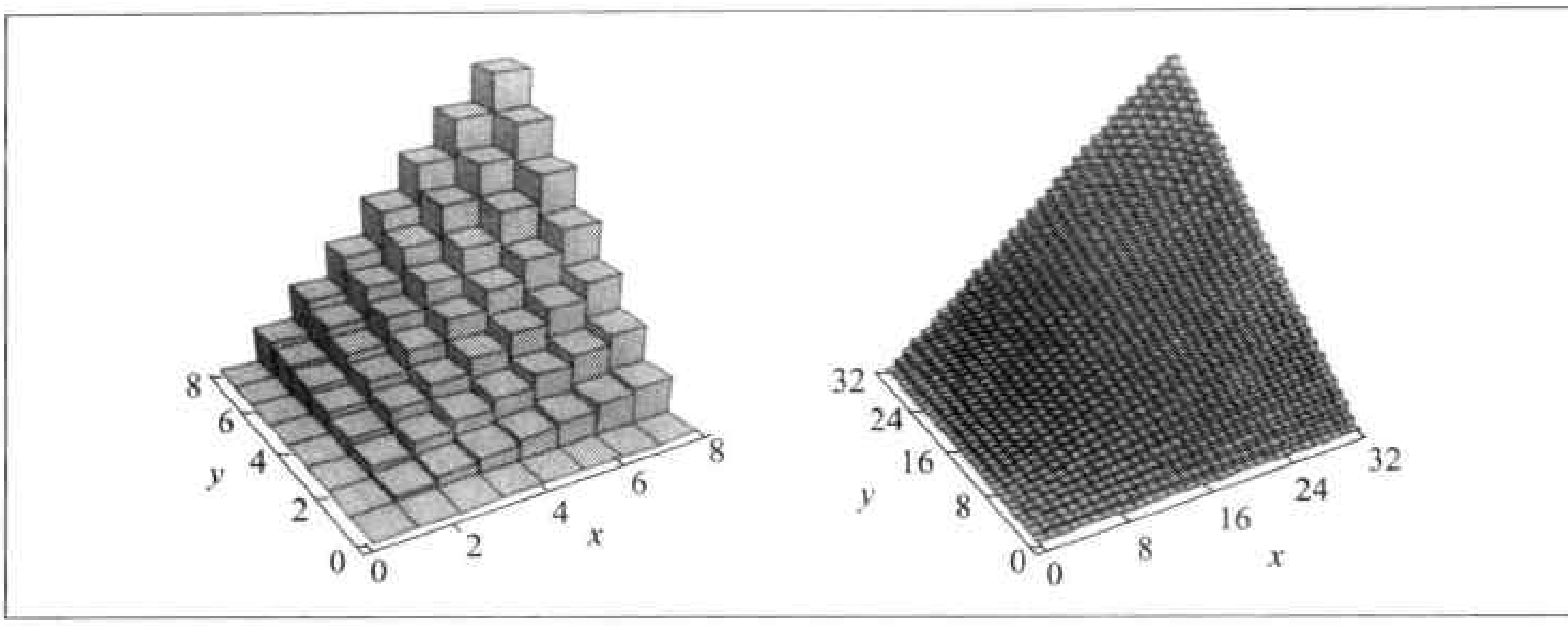


图 C.7　无符号数 x 与 y 之积的图像

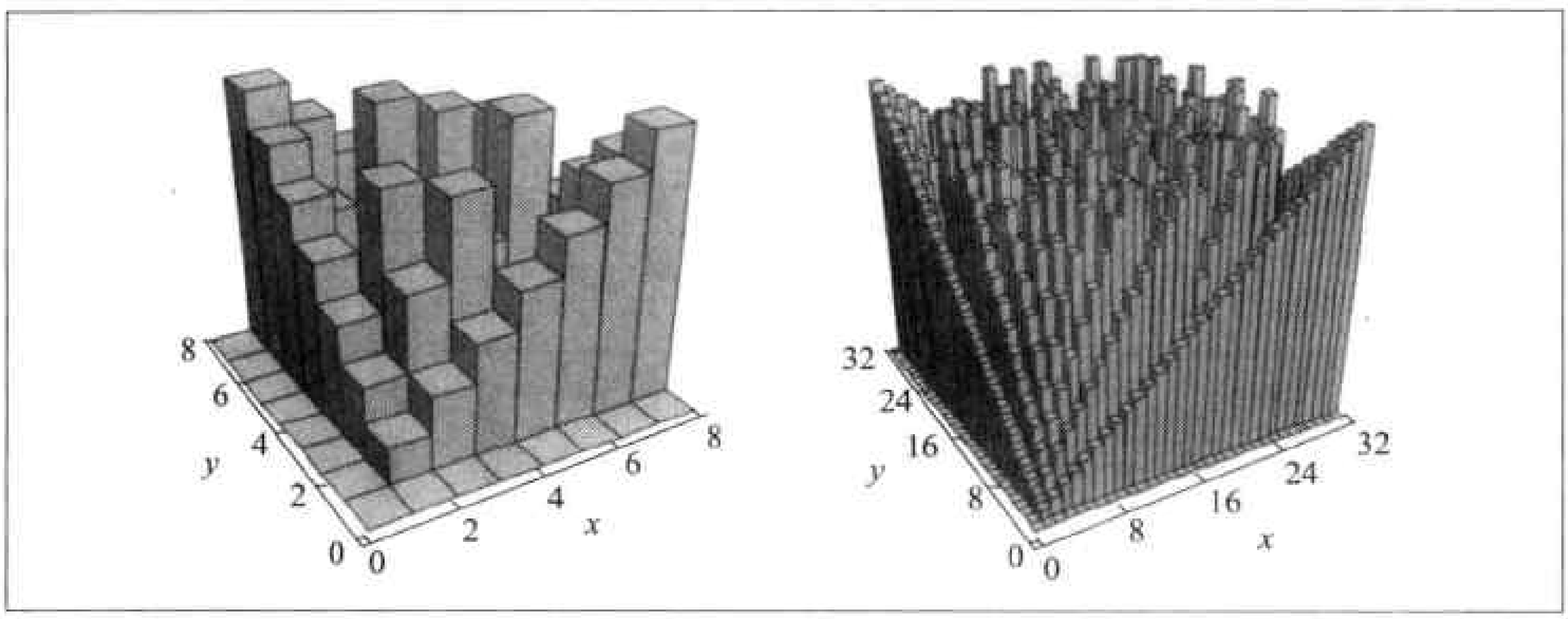


图 C.8　无符号数 x 与 y 之“低阶半积”(low-order half of the product) 的图像

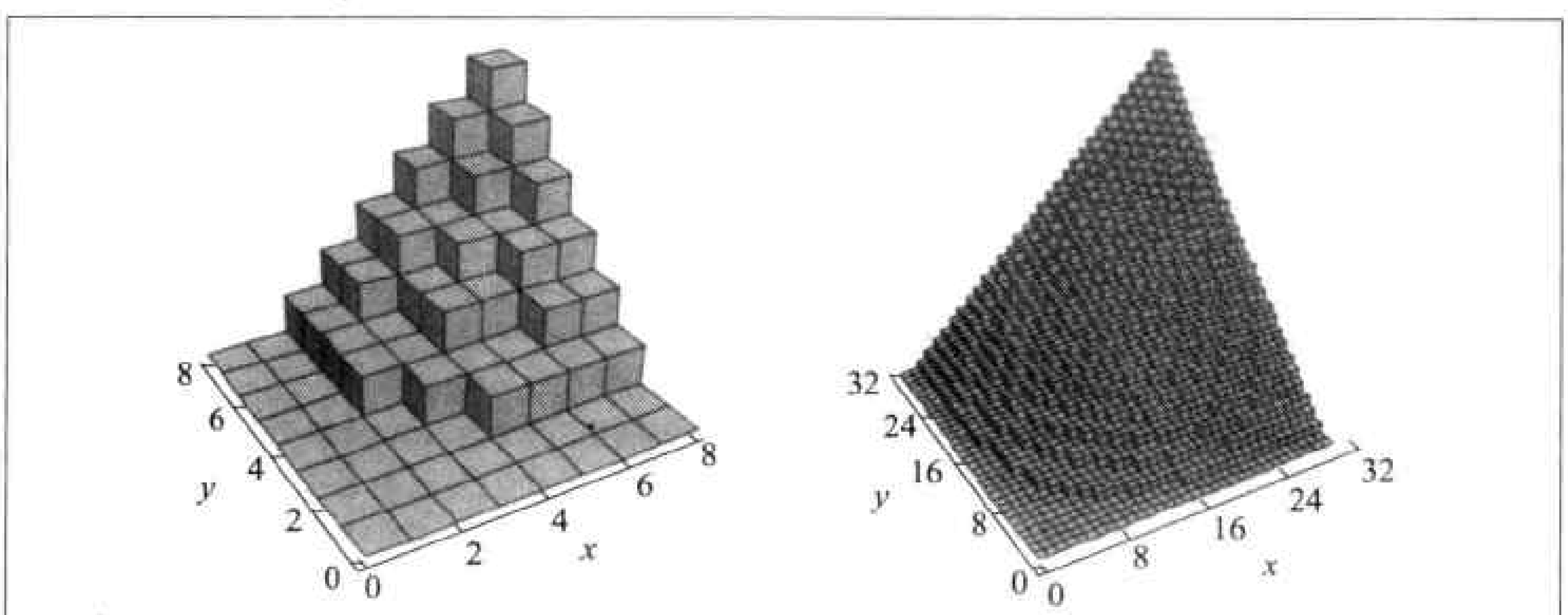


图 C.9　无符号数 x 与 y 之“高阶半积”(high-order half of the product) 的图像

C.3 与除法相关的函数的图像

本节列出了求商函数、求余函数、最大公约数（Greatest Common Divisor，GCD）函数、最小公倍数（Least Common Multiple，LCM）函数的三维图像，其变量 x、y 为非负整数，四者分别如图 C.10、图 C.11、图 C.12 和图 C.13 所示。请注意，图 C.10 的原点位于右远角（rightmost corner）。

图 C.13 压缩了垂直方向上的刻度，在左图中，最高峰的高度为 LCM（6，7）＝42。

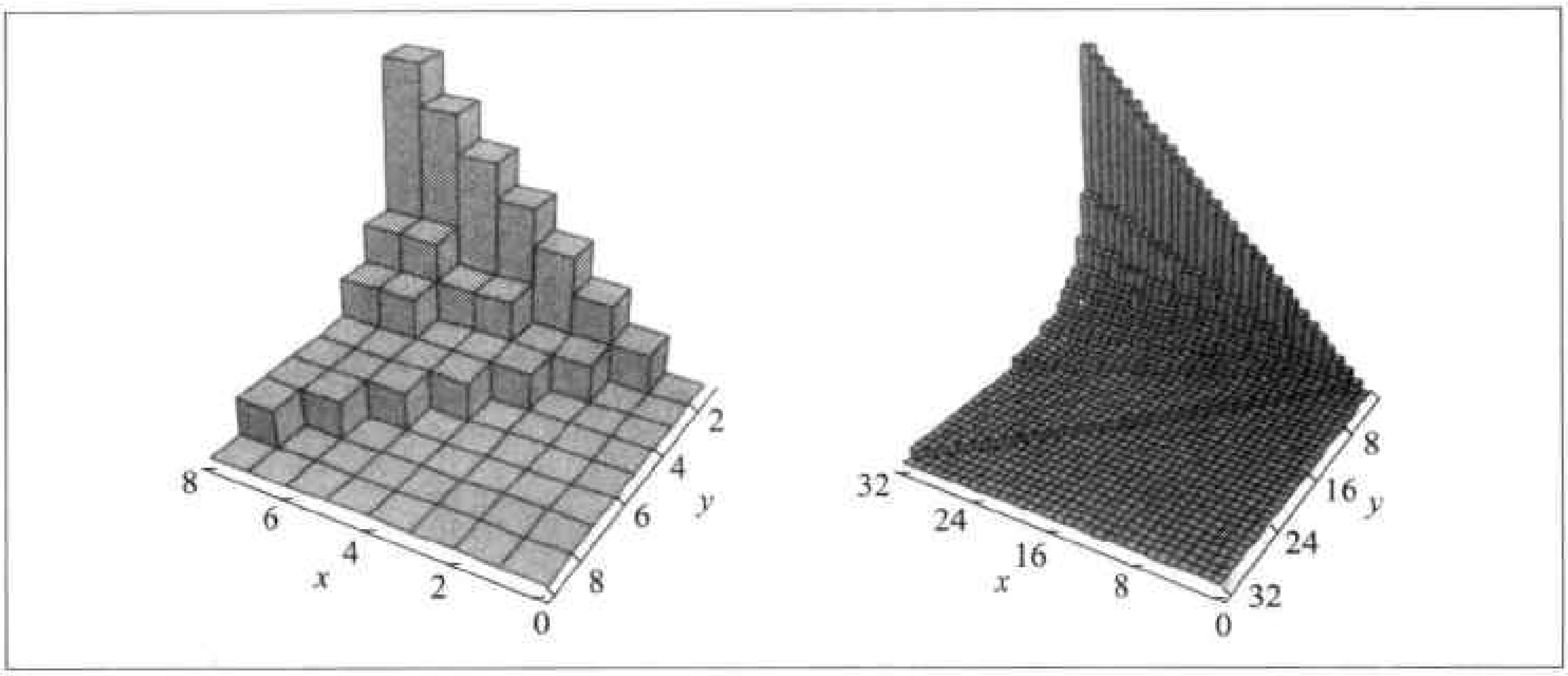


图 C.10 整数求商函数 $x \div y$ 的图像

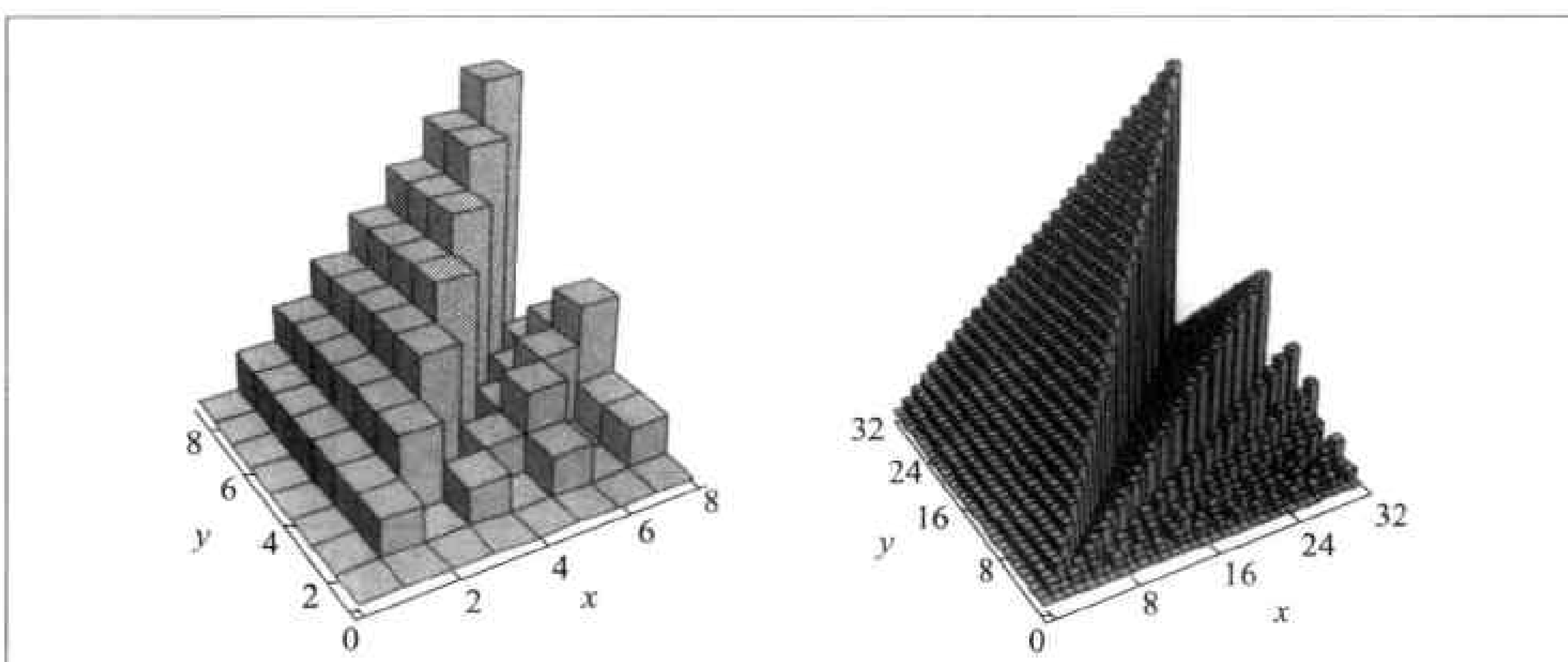


图 C.11 求余函数 rem(x,y) 的图像

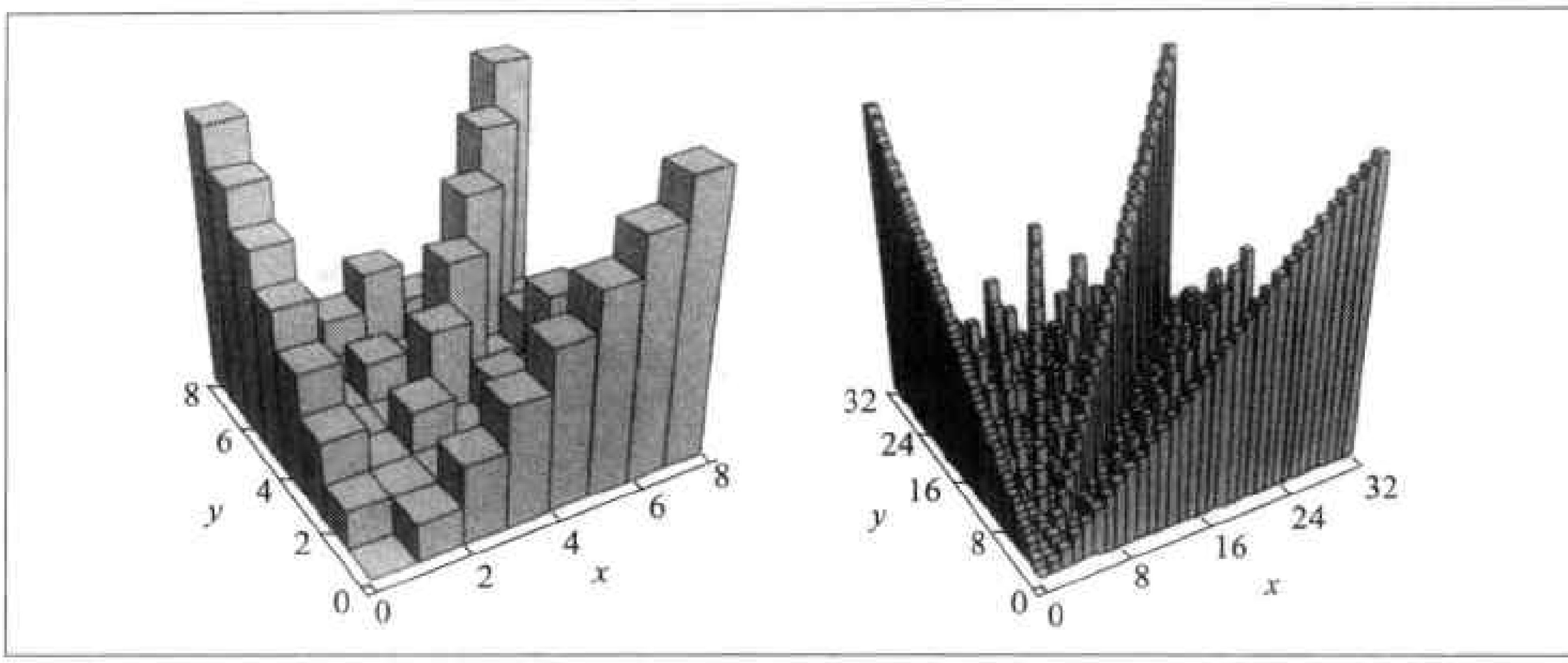


图 C.12 最大公约数函数 GCD（x,y）的图像

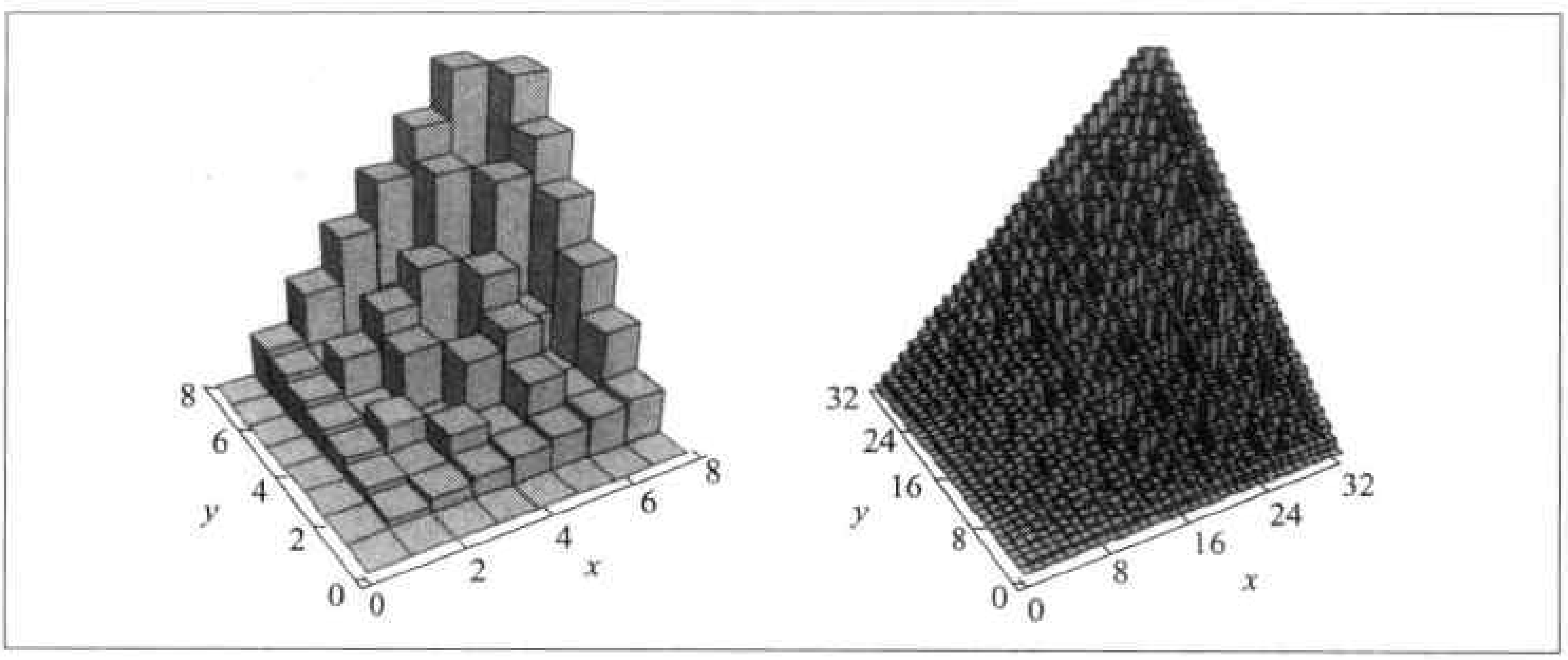


图 C.13 最小公倍数函数 LCM（x,y）的图像

C.4 压缩函数、SAG 函数、循环左移函数的图像

本节列出了压缩函数 compress（$\boldsymbol{x},\boldsymbol{m}$）、SAG（$\boldsymbol{x},\boldsymbol{m}$）、循环左移函数 $x \overset{rot}{\ll} r$ 的三维图像，前两者的参数是整数 $\boldsymbol{x}$ 与 $\boldsymbol{m}$，后者的参数是整数 $\boldsymbol{x}$ 与 $\boldsymbol{r}$，图 C.14、图 C.15、图 C.16 分别描述了这三个函数。

在压缩函数和 SAG 函数（Sheep and Goats，又称绵羊山羊分离函数、分羊函数）中，$\boldsymbol{m}$ 是掩码。压缩函数会根据掩码 $\boldsymbol{m}$ 选取 $\boldsymbol{x}$ 中的某些位元，然后把提取出来的位元压缩至目标字组右侧，左方以 0 填充。SAG 函数则会根据 $\boldsymbol{m}$ 从 $\boldsymbol{x}$ 中选取位元，把选中的位元压缩至目标字组左侧，把未选中的位元压缩至目标字组右侧。

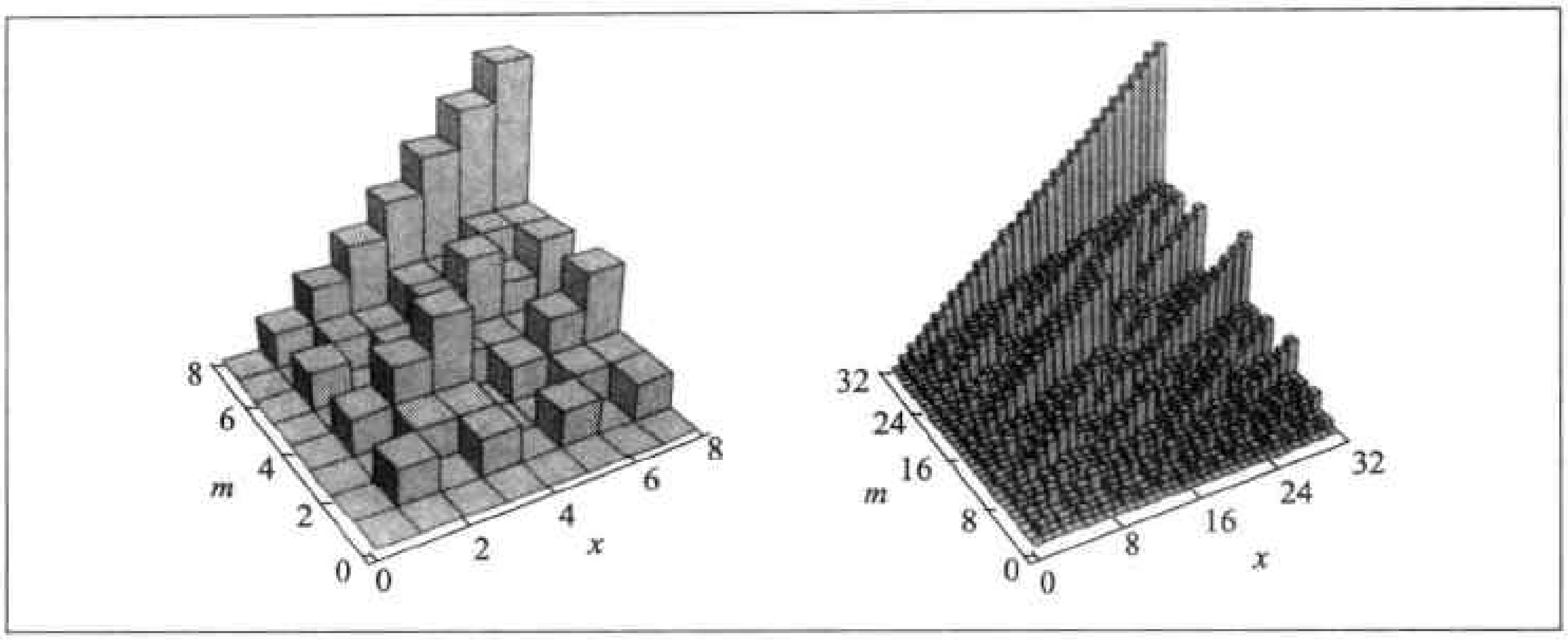


图 C. 14　广义提取函数 compress（x,m）的图像

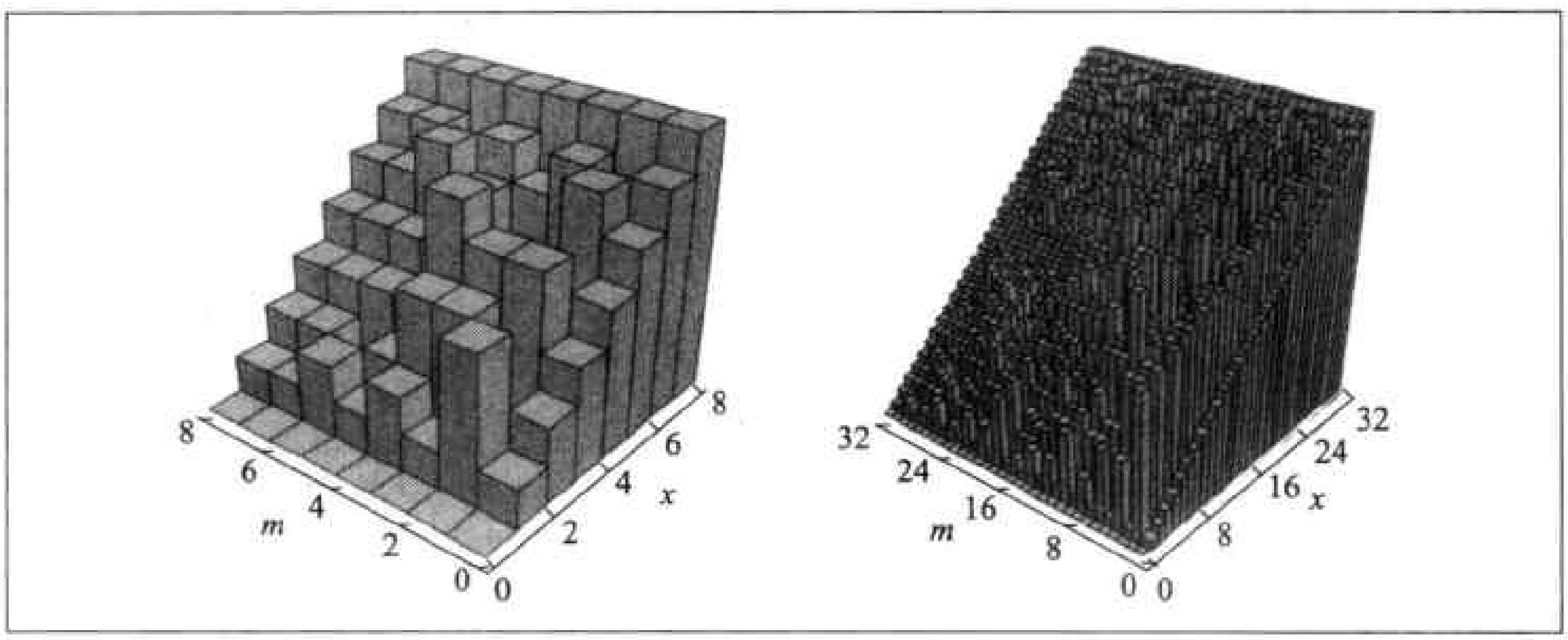


图 C. 15　绵羊山羊分离函数 SAG（x,m）的图像

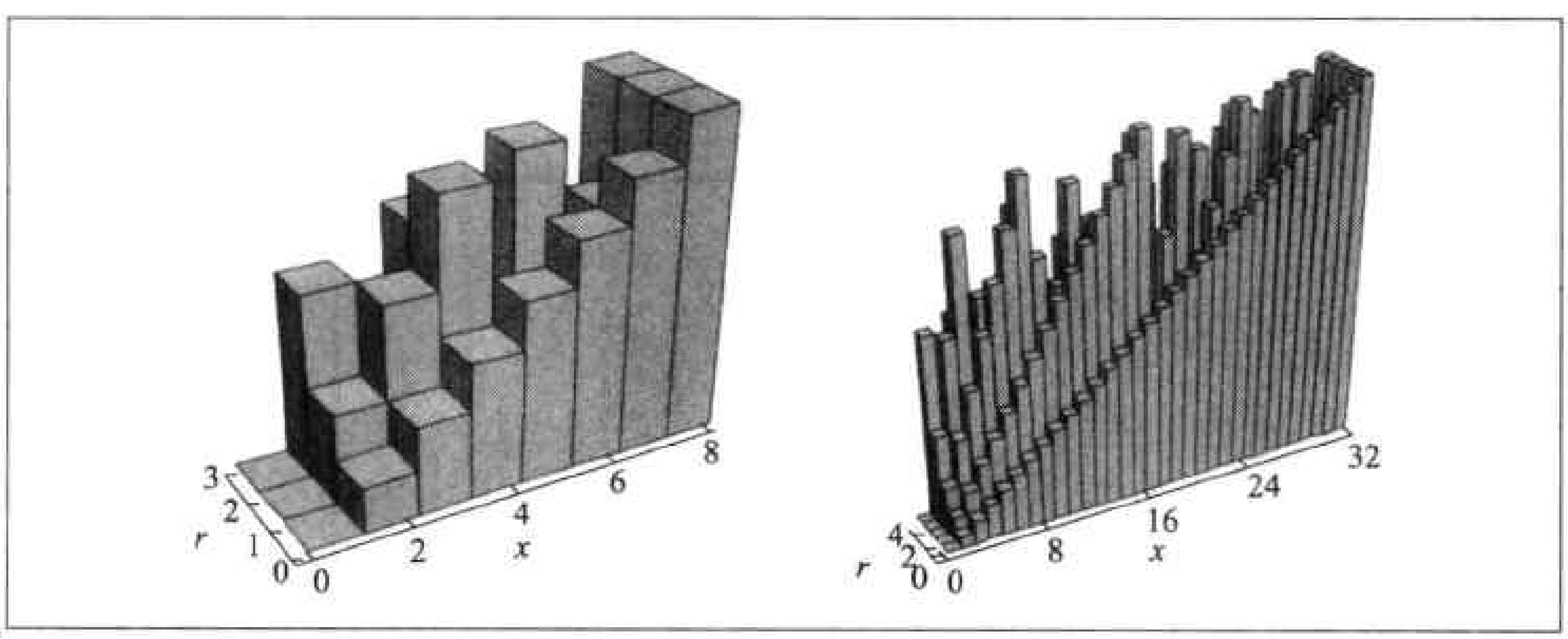


图 C. 16　循环左移函数 $x \overset{rot}{\ll} r$ 的图像

C.5　某些一元函数的二维图像

图 C.17 至图 C.21 列出了某些一元函数的二维图像，其中已将相关函数所操作的位串转化为对应整数了。与三维图像类似，这些图也是由 Mathematica 生成。大部分函数都对应于两张图像：一张描述字长为 4 位时的情况，另一张描述字长为 7 位时的情况。

"格雷码函数"（Gray code function）可以把表示位移量或旋转量的整数映射为与其所表示的量相对应的格雷码。而"逆格雷码函数"（inverse Gray code function）则把格雷码映射成对应的位移量或旋转量。

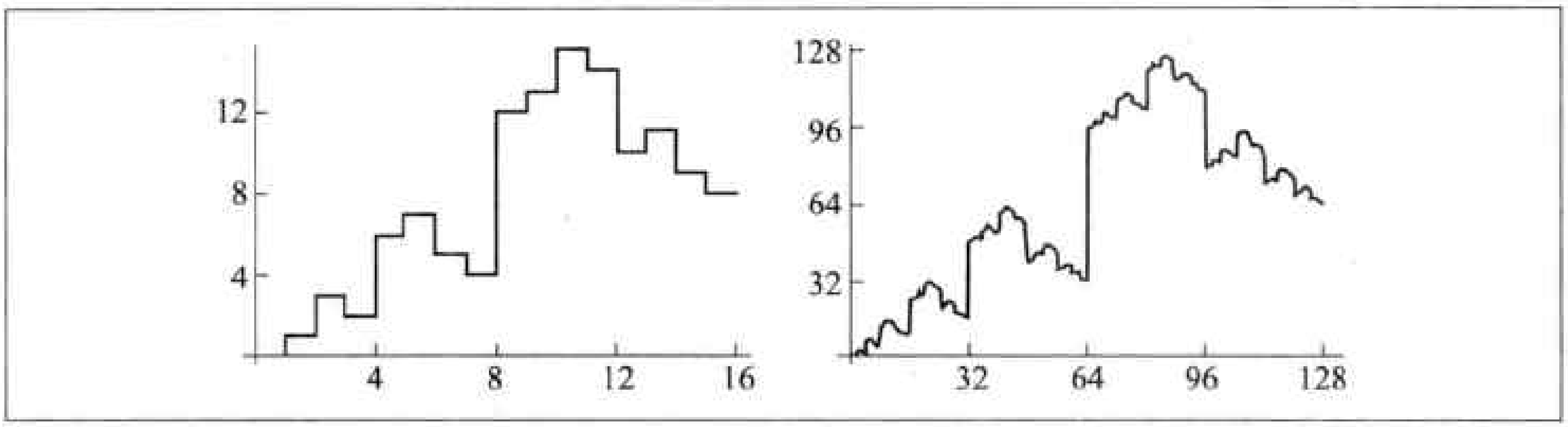


图 C.17　格雷码函数的图像

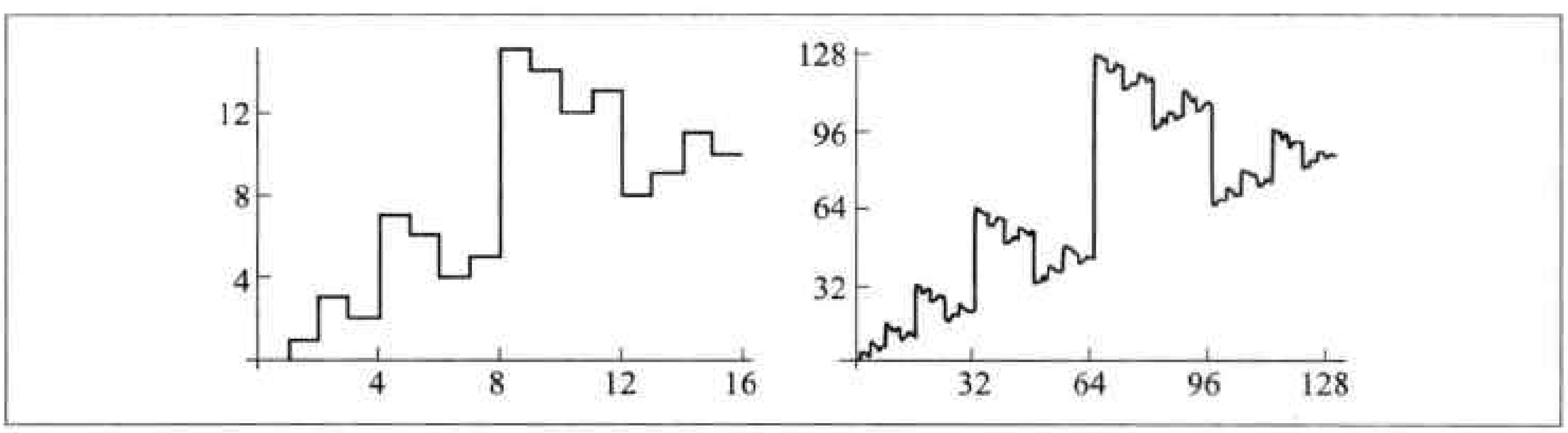


图 C.18　逆格雷码函数的图像

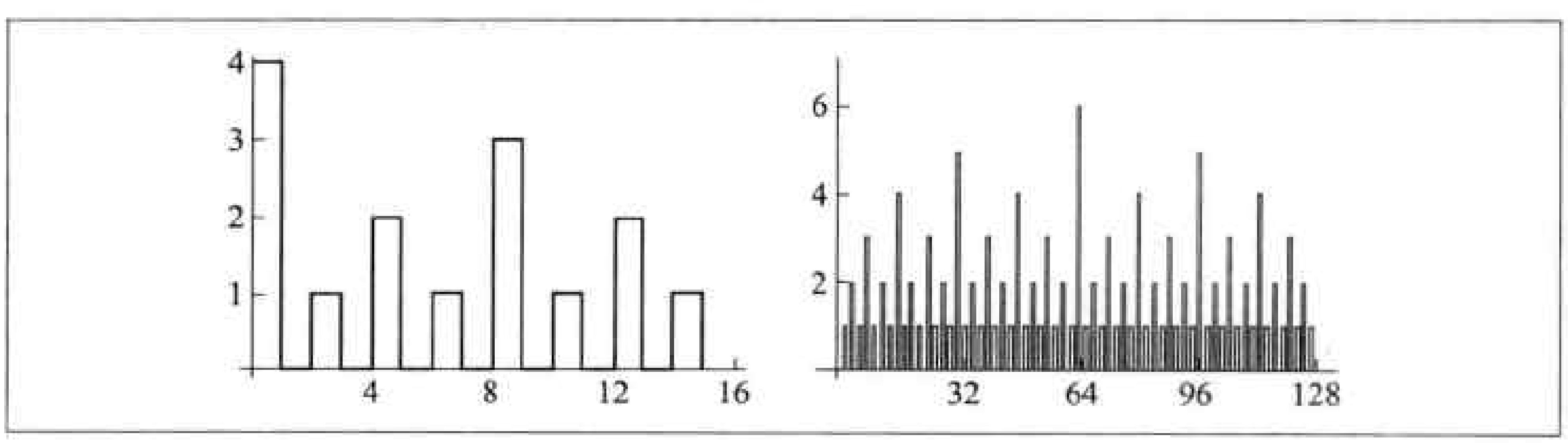


图 C.19　"标尺函数"（其函数值为参数二进制形式中后缀 0 的个数）的图像

假设有一叠卡牌，共 16 张，将其编为 0 至 15 号，那么图 C.22 就表示经过 1 次、两次、3 次"完美外洗牌"（这种洗牌方式能保证首尾两张牌在每次洗完牌后位置不变）之

后这 16 张牌现在的位置。x 坐标表示某张牌的初始位置，而 y 坐标则表示该张牌历经 1 次、两次、3 次洗牌之后的最终位置。图 C.23 也描述了经过 1 轮、两轮、3 轮洗牌之后这 16 张牌的最终位置，不过执行的是“完美内洗牌”。图 C.24 与图 C.25 分别演示了前两者的逆操作。

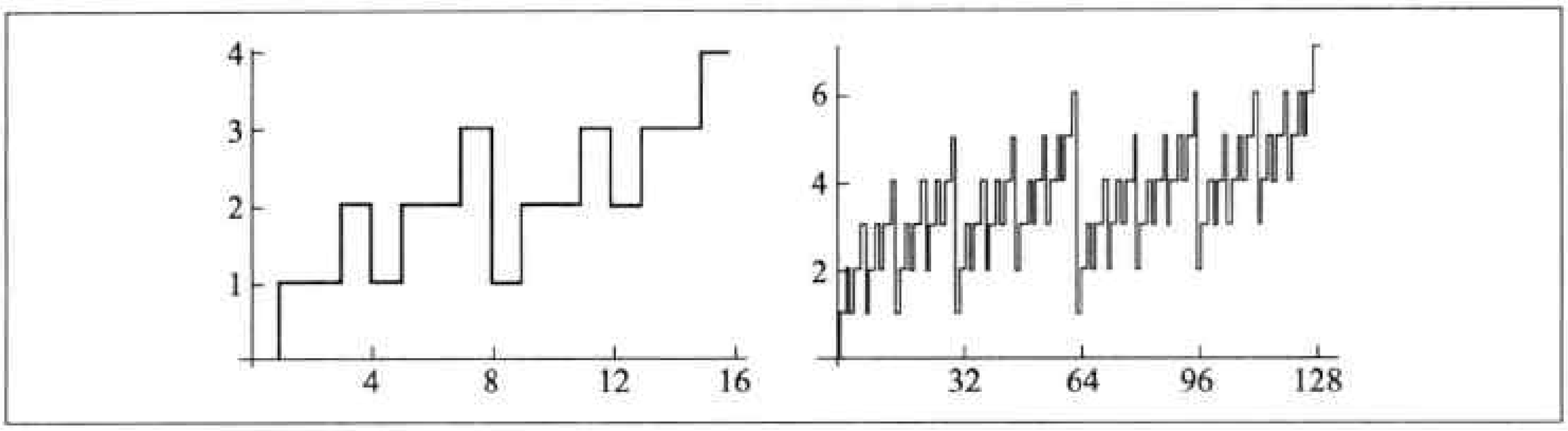


图 C.20　种群计数函数（统计其参数的二进制形式中有多少个值为“1”的位元）的图像

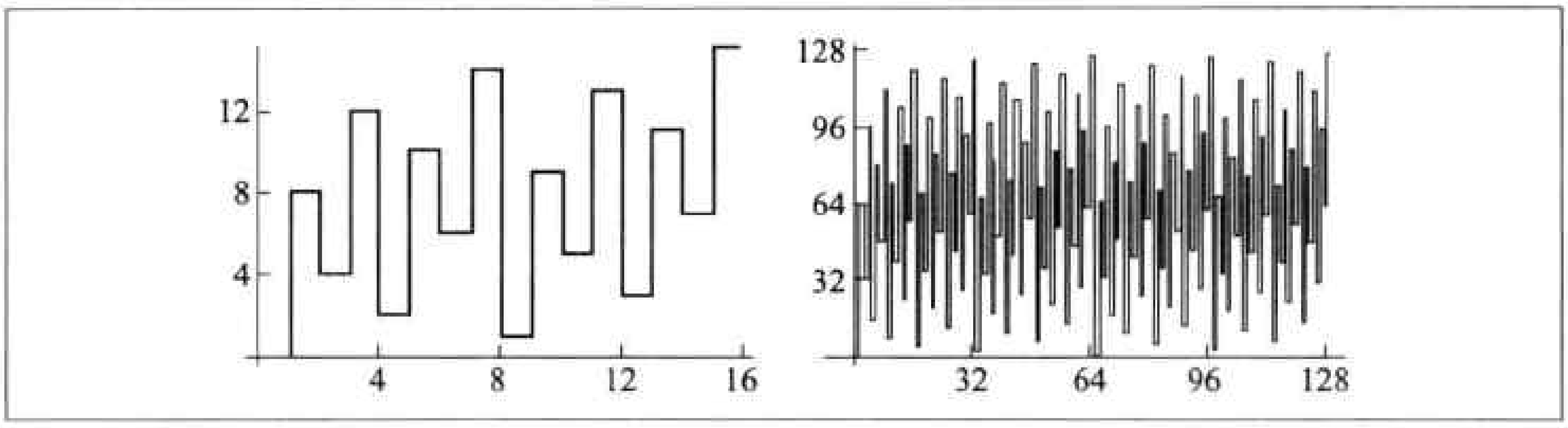


图 C.21　位元反转函数（bit reversal function）的图像

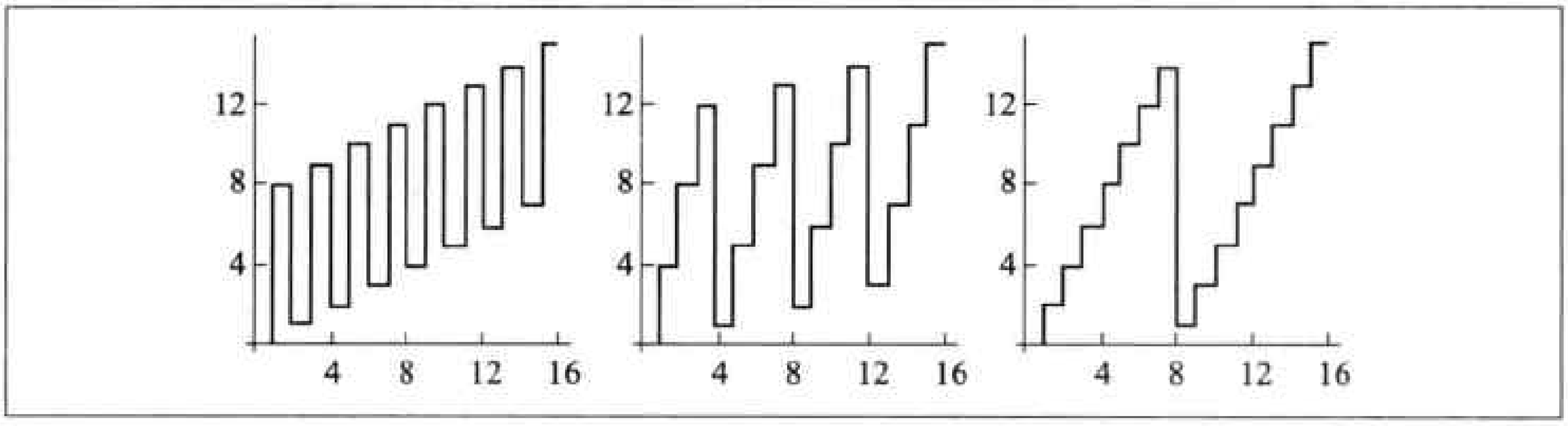


图 C.22　“完美外洗牌”（outer perfect shuffle）函数的图像

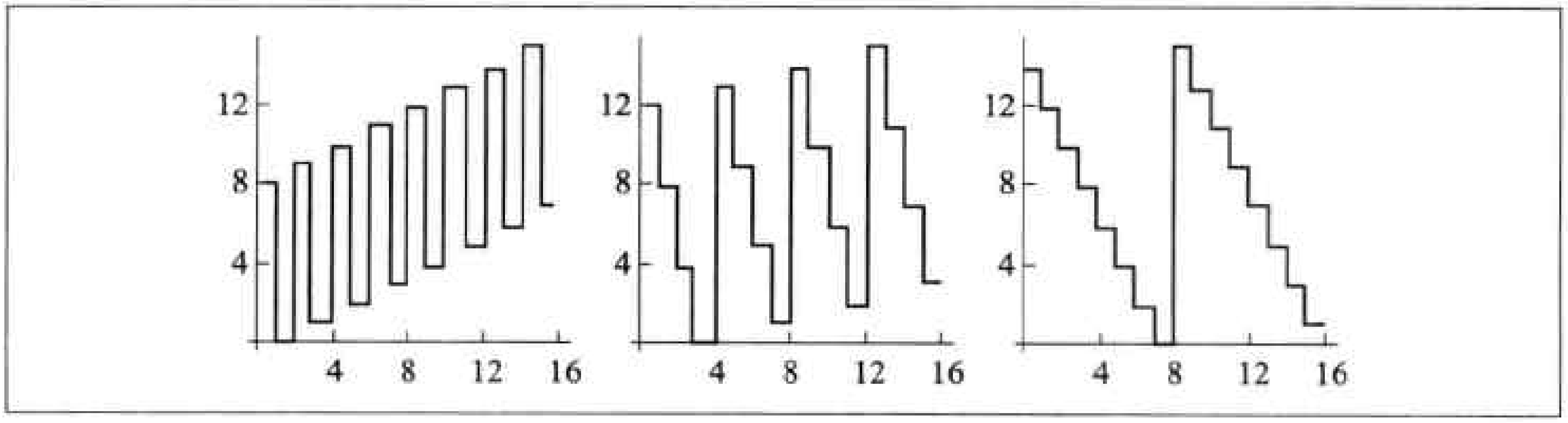


图 C.23　“完美内洗牌”（inner perfect shuffle）函数的图像

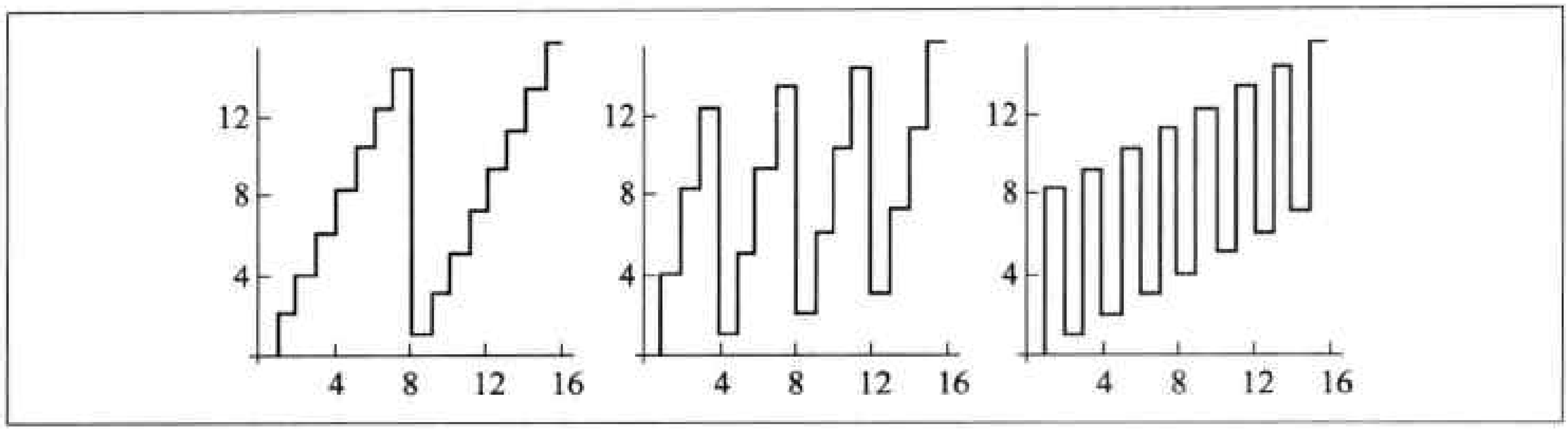


图 C.24 “完美外理牌”（outer perfect unshuffle）函数的图像

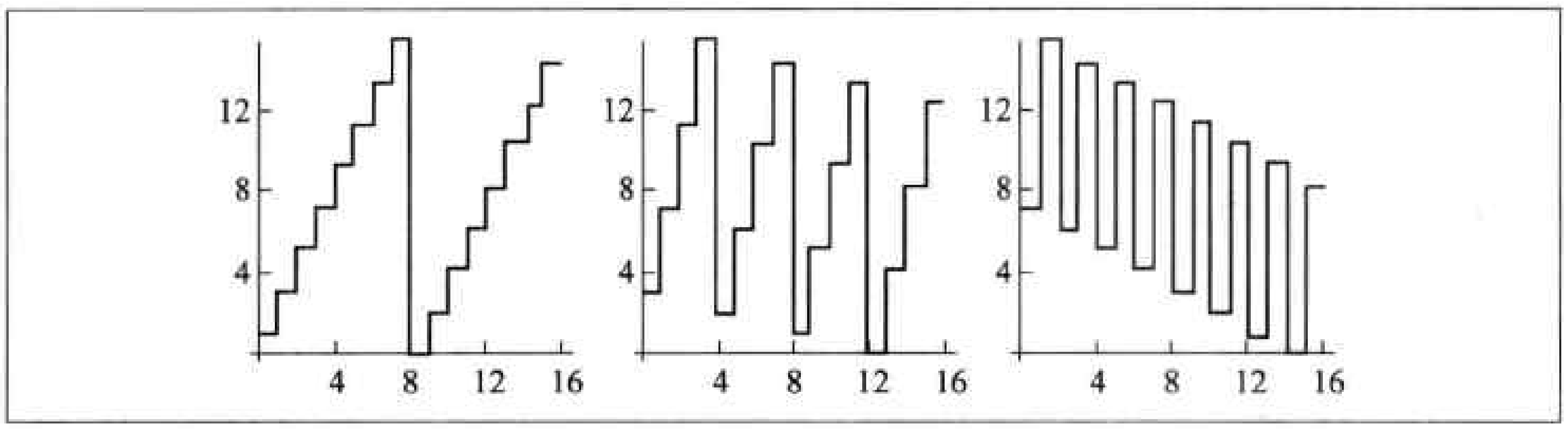


图 C.25 “完美内理牌”（inner perfect unshuffle）函数的图像

图 C.26 与图 C.27 分别演示了对 4 位整数和 8 位整数中的二进制位执行“洗牌”操作后的结果。用不那么严谨的话来说，这两个函数可以定义为：

$$\text{shuffleBits}(x)=\text{asInteger}(\text{shuffle}(\text{bits}(x)))$$

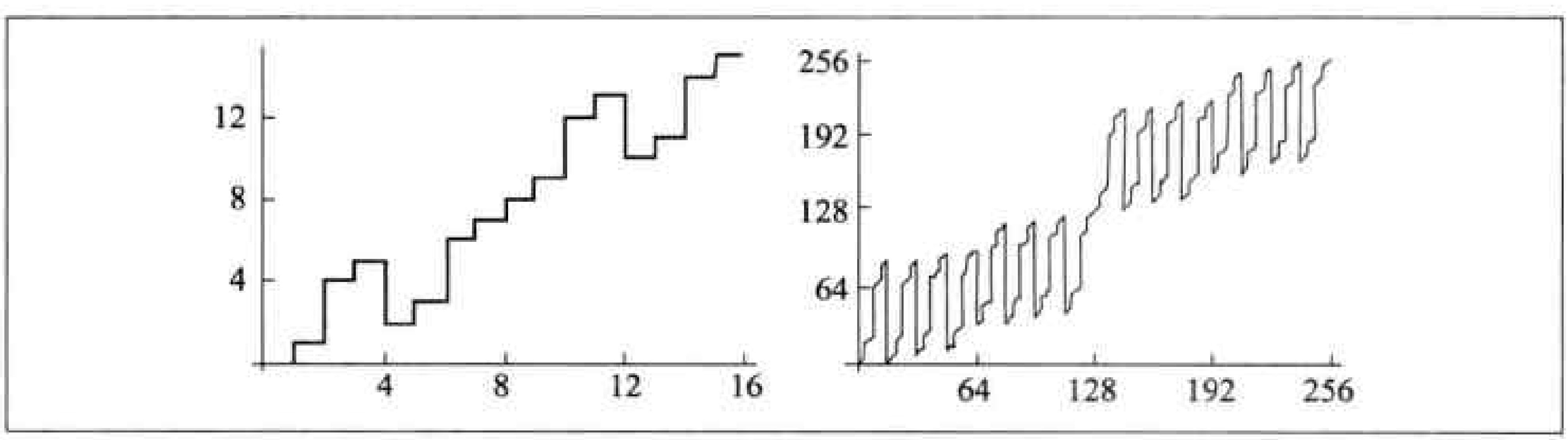


图 C.26 “二进制位完美外洗牌”（outer perfect shuffle bits）函数的图像

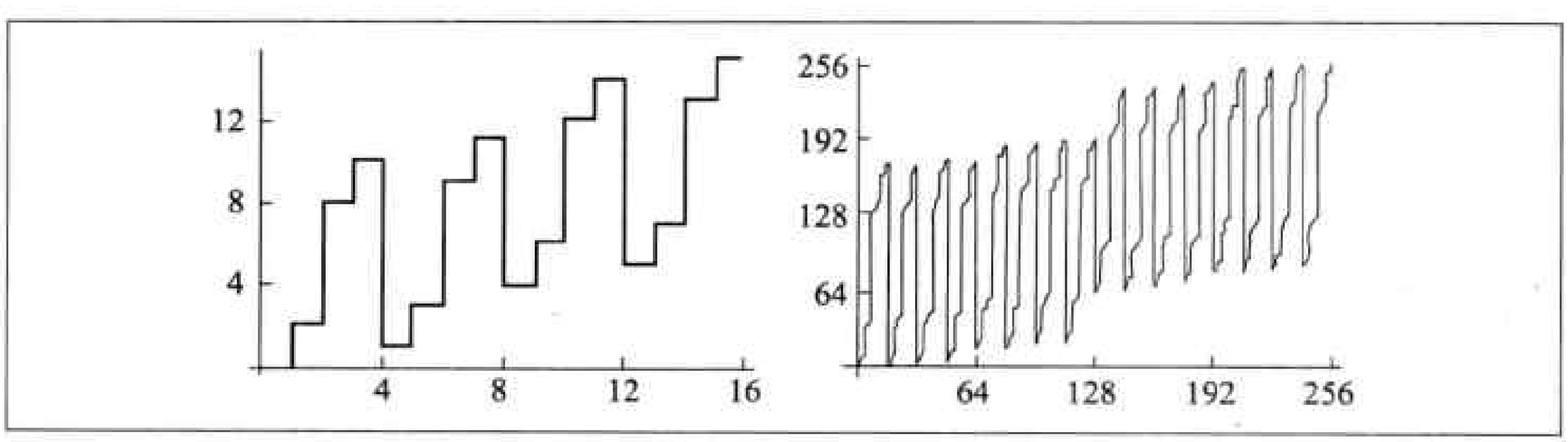


图 C.27 “二进制位完美内洗牌”（inner perfect shuffle bits）函数的图像

参考文献

[AES] *Advanced Encryption Standard* (AES), National Institute of Standards and Technology, FIPS PUB 197 (November 2001). 可于如下网址查阅：http：//csrc. nist. gov/publications/fips/fips197/fips-197. pdf。

[Agrell] Agrell, Erik. http：//webfiles. portal. chalmers. se/s2/research/kit/bounds/，最近一次更新于 2004 年 7 月。

[Allen] 与 Allen，Joseph H. 的私人通信。

[Alv] Alverson, Robert. "Integer Division Using Reciprocals." In *Proceedings IEEE 10th Symposium on Computer Arithmetic*, June 26-28, 1991, Grenoble, France, 186-190.

[Arndt] Arndt, Jörg. *Matters Computational*: *Ideas*, *Algorithms*, *Source Code*. Springer-Verlag, 2010. 该书电子版网址：http：//www. jjj. de/fxt/ #fxtbook。

[Aus1] 得自 Marc A. Auslander 所写的 REXX[⊖] 解释器子程序代码。

[Aus2] 与 Auslander，Marc A. 的私人通信。

[Baum] 高德纳（D. E. Knuth）称，此"三元算法"得自一本未公开发行的技术手册。它由 Bruce Baumgart 于 20 世纪 70 年代中期所编，其中比较了 PDP10 系列计算机所用的大约 20 种位元反转算法。

[Bern] Bernstein, Robert. "Multiplication by Integer Constants." *Software—Practice and Experience* 16, 7 (July 1986), 641-652.

[BGN] Burks, Arthur W., Goldstine, Herman H., and von Neumann, John. "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, Second Edition" (1947). In *Papers of John von Neumann on Computing and Computing Theory*, Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987.

⊖ Restructured Extended Executor，1979 年由 IBM 研发的程序设计语言，主要用在 IBM 大型计算机上，其他平台也有其解释器或编译器。详情参见：https：//zh. wikipedia. org/wiki/REXX。——译者注

[Black] Black, Richard. Web site www. cl. cam. ac. uk/Research/SRG/bluebook/21/crc/crc. html. University of Cambridge Computer Laboratory Systems Research Group, February 1994.

[Bonz] 与 Bonzini，Paolo 的私人通信。

[Brou] Brouwer, Andries E. http://www. win. tue. nl/~ aeb/codes/binary-1. html，最近一次更新于 2012 年 1 月。

[CavWer] Cavagnino, D. and Werbrouck, A. E. "Efficient Algorithms for Integer Division by Constants Using Multiplication." *The Computer Journal* 51, 4 (2008), 470-480.

[CC] Caldwell, Chris K. and Cheng, Yuanyou. "Determining Mills' Constant and a Note on Honaker's Problem." *Journal of Integer Sequences* 8, 4 (2005), article 05. 4. 1, 9 pp. 该文亦可于此网址查阅：http://www. cs. uwaterloo. ca/journals/JIS/VOL8/Caldwell/caldwell78. pdf。

[CJS] 与 Stephenson，Christopher J. 的私人通信。

[Cohen] Norman H. Cohen 指出了代码中与 ANSI 标准规则相违背之处。

[Cplant] Leung, Vitus J., et. al. "Processor Allocation on Cplant: Achieving General Processor Locality Using One-Dimensional Allocation Strategies." In *Proceedings 4th IEEE International Conference on Cluster Computing*, September 2002, 296-304.

[Cut] Cutland, Nigel J. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[CWG] Hoxey, Karim, Hay, and Warren (Editors). *The PowerPC Compiler Writer's Guide*. Warthman Associates, 1996.

[Dalton] 与 Dalton，Michael 的私人通信。

[Danne] 与 Dannemiller，Christopher M. 的私人通信。他从 Linux 源码库（www. gelato. unsw. edu. au/lxr/source/lib/crc32. c，第 105 至 111 行）中得知此做法。

[DES] Data Encryption Standard (DES), National Institute of Standards and Technology, FIPS PUB 46-2 (December 1993). 可于如下网址查阅：http://www. itl. nist. gov/fipspubs/fip46-2. htm。

[Dewd] Dewdney, A. K. *The Turing Omnibus*. Computer Science Press, 1989.

[Dietz] Dietz, Henry G. http://aggregate. org/MAGIC/

[Ditlow] 与 Ditlow，Gary S. 的私人通信。

[Dubé] Dubé，Danny 于 1997 年 10 月 3 日在新闻组 comp. compression. research 所发帖子。

[Dud] Dudley, Underwood. "History of a Formula for Primes." *American Mathematics*

Monthly 76 (1969), 23-28.

[EL] Ercegovac, Miloš D. and Lang, Tomás. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994.

[Etzion] Etzion, Tuvi. "Constructions for Perfect 2-Burst-Correcting Codes," *IEEE Transactions on Information Theory* 47, 6 (September 2001), 2553-2555.

[Floyd] Floyd, Robert W. "Permuting Information in Idealized Two-Level Storage." In *Complexity of Computer Computations* (Conference proceedings), Plenum Press, 1972, 105-109. 据笔者所知，它是最早提到这种 $2^n \times 2^n$ 矩阵转置法的参考资料。

[Gard] Gardner, Martin. "Mathematical Games" column in *Scientific American* 227, 2 (August 1972), 106-109.

[Gaud] 与 Gaudet, Dean 的私人通信。

[GGS] Gregoire, Dennis G., Groves, Randall D., and Schmookler, Martin S. *Single Cycle Merge/Logic Unit*, US Patent No. 4, 903, 228, February 20, 1990.

[GK] Granlund, Torbjörn and Kenner, Richard. "Eliminating Branches Using a Superoptimizer and the GNU C Compiler." In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), July 1992, 341-352.

[GKP] Graham, Ronald L., Knuth, Donald E., and Patashnik, Oren. Concrete Mathematics: *A Foundation for Computer Science*, *Second Edition*. Addison-Wesley, 1994.
《具体数学——计算机科学基础（英文版·第2版）》，机械工业出版社，2008年。

[GLS1] 与 Steele, Guy L., Jr. 的私人通信。

[GLS2] Steele, Guy L., Jr. "Arithmetic Shifting Considered Harmful." AI Memo 378, MIT Artificial Intelligence Laboratory (September 1976); also in *SIGPLAN Notices* 12, 11 (November 1977), 61-69.

[GM] Granlund, Torbjörn and Montgomery, Peter L. "Division by Invariant Integers Using Multiplication." In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (PLDI), August 1994, 61-72.

[Gold] 第二个表达式由 Richard Goldberg 给出。

[Good] Goodstein, Prof. R. L. "Formulae for Primes." *The Mathematical Gazette* 51 (1967), 35-36.

[Gor] 与 Goryavsky，Julius 的私人通信。

[GSO] GNU Superoptimizer 所用的方法。

[HAK] Beeler，M.，Gosper，R. W.，and Schroeppel，R. HAKMEM，MIT Artificial Intelligence Laboratory AIM 239，February 1972.

[Ham] Hamming，Richard W.，"Error Detecting and Error Correcting Codes，" *The Bell System Technical Journal* 26，2（April 1950），147-160.

[Harley] Harley，Robert 于 1996 年 7 月 12 日在新闻组 comp. arch 所发帖子。

[Hay1] 与 Hay，R. W. 的私人通信。

[Hay2] 第 1 个表达式出现在 R. W. Hay 所写的一段编译器子程序中。

[Hil] Hilbert，David. "Ueber die stetige Abbildung einer Linie auf ein Flächenstück." *Mathematischen Annalen* 38（1891），459-460.

[Hill] Hill，Raymond. *A First Course in Coding Theory*. Clarendon Press，1986.

[HilPat] Hiltgen，Alain P. and Paterson，Kenneth G. "Single-Track Circuit Codes." *IEEE Transactions on Information Theory* 47，6（2001）2587-2595.

[Hop] 与 Hopkins，Martin E. 的私人通信。

[HS] Hillis，W. Daniel and Steele，Guy L.，Jr. "Data Parallel Algorithms." *Comm. ACM* 29，12（December 1986）1170-1183.

[Hsieh] Hsieh，Paul 于 2005 年 4 月 29 日在新闻组 comp. lang. c 所发帖子。

[Huef] 与 Hueffner，Falk 的私人通信。

[H&P] Hennessy，John L. and Patterson，David A. *Computer Architecture：A Quantitative Approach*. Morgan Kaufmann，1990.
《计算机体系结构：量化研究方法（英文版·第 5 版）》，机械工业出版社，2012。

[H&S] Harbison，Samuel P. and Steele，Guy L.，Jr. *C：A Reference Manual*，Fourth Edition. Prentice-Hall，1995.
《C 语言参考手册（原书第 5 版）》，机械工业出版社，2004。

[H&W] Hardy，G. H. and Wright，E. M. *An Introduction to the Theory of Numbers*，Fourth Edition. Oxford University Press，1960.

[IBM] 1961 年的 IBM 程序开发课程。

[Irvine] Irvine，M. M. "Early Digital Computers at Bell Telephone Laboratories." *IEEE Annals of the History of Computing* 23，3（July-September 2001），22-42.

[JVN] von Neumann，John. "First Draft of a Report on the EDVAC." In *Papers of John von Neumann on Computing and Computing Theory*，Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing，MIT Press，1987.

[Karat] Karatsuba, A. and Ofman, Yu. "Multiplication of multidigit numbers on automata." *Soviet Physics-Doklady* 7, 7 (January 1963), 595-596. 两位作者从理论上证实了 m 位二进制整数相乘的算法复杂度是 $O(m^{\log_2 3}) \approx O(m^{1.585})$，不过与高斯通过3次乘法计算两复数之积相比，他们用的方法更为繁复。

[Karv] Karvonen, Vesa. 得自"The Assembly Gems"网页：http://www.df.lth.se/~john_e/fr_gems.html。

[Keane] Keane, Joe 于1995年7月9日在新闻组 sci.math.num-analysis 所发帖子。

[Knu1] Knuth, Donald E. *The Art of Computer Programming, Volume 1, Third Edition: Fundamental Algorithms*. Addison-Wesley, 1997.
《计算机程序设计艺术 第1卷 基本算法（英文版·第3版）》，机械工业出版社，2008年。

[Knu2] Knuth, Donald E. *The Art of Computer Programming, Volume 2, Third Edition: Seminumerical Algorithms*. Addison-Wesley, 1998.
《计算机程序设计艺术 第2卷 半数值算法（英文版·第3版）》，机械工业出版社，2008年。

[Knu3] 对于以负整数做数制底数来执行算术这一想法，许多人都曾独立提出过。据高德纳所述，Vittorio Grünwald[⊖]在1885年最早提出此思路。高德纳本人也曾在1955年向"高中生科学奖"（"science talent search" for high-school seniors）提交了一篇关于该主题的论文。早期参考资料请见[Knu2]。

[Knu4] Knuth, Donald E. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*, Section 7.1.1. Addison-Wesley, 2011.

[Knu5] 前书 Section 7.1.3。作者高德纳说：判断前导0个数是否相等的关系式是由 W. C. Lynch 在2006年给出的。

[Knu6] 前书 Section 7.2.1.1，习题80。

[Knu7] Knuth, Donald E. *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX-A RISC Computer for the New Millennium*. Addison-Wesley, 2005.
《计算机程序设计艺术（第1卷第1册双语版）——MMIX：新千年的 RISC 计算机》，苏运霖译，机械工业出版社，2006。

[Knu8] 与高德纳的私人通信。

[KRS] Kruskal, Clyde P., Rudolph, Larry, and Snir, Marc. "The Power of Parallel Prefix." *IEEE Transactions on Computers* C-34, 10 (October 1985), 965-968.

⊖ Vittorio Grünwald（1855—1943），意大利人，数学教授、德语教授。详情参见：http://en.wikipedia.org/wiki/Vittorio_Gr%C3%BCnwald。——译者注

[Kumar] 此图根据 Gowri Kumar 在私人通信中的提议而增补。

[Lamp] Lamport, Leslie. "Multiple Byte Processing with Full-Word Instructions." *Communications of the ACM* 18, 8 (August 1975), 471-475.

[Lang] Langdon, Glen G. Jr., "Subtraction by Minuend Complementation," *IEEE Transactions on Computers* C-18, 1 (January 1969), 74-76.

[LC] Lin, Shu and Costello, Daniel J., Jr. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983.

[Lomo] Lomont, Chris. *Fast Inverse Square Root*. www.lomont.org/Math/Papers/2003/InvSqrt.pdf.

[LPR] Leiserson, Charles E., Prokop, Harald, and Randall, Keith H. *Using de Bruijn Sequences to Index a 1 in a Computer Word*. MIT Laboratory for Computer Science, July 7, 1998.
本文亦可于如下网址查阅：http://supertech.csail.mit.edu/papers/debruijn.pdf。

[LSY] Lee, Ruby B., Shi, Zhijie, and Yang, Xiao. "Efficient Permutation Instructions for Fast Software Cryptography." *IEEE Micro* 21, 6 (November/December 2001), 56-69.

[L &S] Lam, Warren M. and Shapiro, Jerome M. "A Class of Fast Algorithms for the Peano-Hilbert Space-Filling Curve." In *Proceedings ICIP* 94, 1 (1994), 638-641.

[MD] 与 Denneau, Monty M. 的私人通信。

[MIPS] Kane, Gerry and Heinrich, Joe. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[MM] Morton, Mike. "Quibbles & Bits." *Computer Language* 7, 12 (December 1990), 45-55.

[Möbi] 与 Möbius, Stefan K. 的私人通信。

[MS] MacWilliams, Florence J. and Sloane, Neil J. A. *The Theory of Error-Correcting Codes, Part II*. North-Holland, 1977.

[Mycro] Mycroft, Alan 于 1987 年 4 月 8 日在新闻组 comp.arch 所发帖子。

[Neum] 与 Neumann, Jasper L. 的私人通信。

[NZM] Niven, Ivan, Zuckerman, Herbert S., and Montgomery, Hugh L. *An Introduction to the Theory of Numbers, Fifth Edition*. John Wiley & Sons, Inc., 1991.

[PeBr] Peterson, W. W. and Brown, D. T. "Cyclic Codes for Error Detection." In *Proceedings of the IRE*, 1 (January 1961), 228-235.

[PHO] 与 Oden, Peter H. 的私人通信。

[PL8] 笔者从 PL.8 编译器[⊖]中得知此技巧。

[PuBr] Purdom，Paul Walton Jr.，and Brown，Cynthia A. *The Analysis of Algorithms*. Holt，Rinehart and Winston，1985.

[Reiser] Reiser，John 于 1998 年 12 月 11 日在新闻组 comp.arch.arithmetic 所发帖子。

[Rib] Ribenboim，Paulo. *The Little Book of Big Primes*. Springer-Verlag，1991.《博大精深的素数（原书第2版）》，孙淑玲，冯克勤译，科学出版社，2007年。

[RND] Reingold，Edward M.，Nievergelt，Jurg，and Deo，Narsingh. *Combinatorial Algorithms：Theory and Practice*. Prentice-Hall，1977.

[Roman] Roman，Steven. *Coding and Information Theory*. Springer-Verlag，1992.

[Sagan] Sagan，Hans. *Space-Filling Curves*. Springer-Verlag，1994. 强烈推荐这本好书，即便你只对此话题略感兴趣也不妨一看。

[Seal1] Seal，David 于 1997 年 5 月 13 日在新闻组 comp.arch.arithmetic 所发帖子。Seal 称：Harley 是据他所知最早将 CSA 运用于此问题的人。Seal 演示了一种非常好的算法，能够在一个很大的数组中统计值为"1"的位元数（如图 5.8 与图 5.9 所示），并且还将该算法运用于一个长度为 7 的数组上（与图 5.10 所示方案类似）。

[Seal2] Seal，David 于 1994 年 2 月 16 日在新闻组 comp.sys.acorn.tech 所发帖子。

[Shep] 与 Shepherd，Arvin D. 的私人通信。

[Stall] Stallman，Richard M. *Using and Porting GNU CC*. Free Software Foundation，1998.

[Strach] Strachey，Christopher. "Bitwise Operations." *Communications of the ACM* 4，3（March 1961），146. 该期杂志还刊登了另外一篇文章（"Two Methods for Word Inversion on the IBM 709，" by Robert A. Price and Paul Des Jardins。1961 年 3 月刊 A13 版列有小幅修正），其中又介绍了两种位元反转算法。本书不讨论那两种算法，因为它们用到了 IBM 709 计算机中略显奇特的 CAQ 指令（*Convert by Addition from the MQ*）。该指令根据一系列索引值来查表，并把内存中的数放入累加器。它不属于 RISC 指令集。

[Tanen] Tanenbaum，Andrew S. *Computer Networks*，Second Edition. Prentice Hall，1988.

[Taro] 此程序作者似乎已不可考。Gary Tarolli 是最早开始使用的人，并略微优化了其中的常量，那时他可能供职于 SGI（Silicon Graphics，Inc.，硅谷图形公司）。Tarolli 也推广了这段程序，据其所说，该代码最迟已于 1995 年出现。欲知更多故事，请参阅：http://www.beyond3d.com/content/articles/8/。

⊖ 由 IBM 于 20 世纪 70 年代研发的编译器，详情参见：http://en.wikipedia.org/wiki/PL/I#Special_purpose_and_system_PL.2FI_compilers。——译者注

[Voor] Voorhies, Douglas. "Space-Filling Curves and a Measure of Coherence." *Graphics Gems II*, AP Professional (1991).

[War] Warren, H. S., Jr. "Functions Realizable with Word-Parallel Logical and Two's-Complement Addition Instructions." *Communications of the ACM* 20, 6 (June 1977), 439-441.

[Weg] 据笔者所知，最早提到此算法的参考资料是：Wegner, P. A. "A Technique for Counting Ones in a Binary Computer." *Communications of the ACM* 3, 5 (May 1960), 322.

[Wells] Wells, David. *The Penguin Dictionary of Curious and Interesting Numbers*. Penguin Books, 1997.

[Will] Willans, C. P. "On Formulae for the nth Prime Number." *The Mathematical Gazette* 48 (1964), 413-415.

[Wood] 与 Woodrum, Luther 的私人通信。第 2 个公式没有用字面常量，而且能在 IBM System/370 系列主机上正常运行。

[Wor] Wormell, C. P. "Formulae for Primes." *The Mathematical Gazette* 51 (1967), 36-38.

[Zadeck] 与 Zadeck, F. Kenneth 的私人通信。

Hacker's Delight

(Second Edition)

“这是第一本宣称能讲解计算机算法隐晦细节的书，而且讲得还真不错。我知道的每一条技巧书里都提到了，而且还讲了好多好多我不知道的。不论是在开发程序库或编译器，还是在极力搜求优雅算法，此书都可谓天赐良册，应放在高德纳所著《计算机程序设计艺术》那套书旁边。本书第一版刊印后的10年间，它对我在Sun和Google的工作大有裨益，而第二版所添加新内容亦令我惊羡不已。”

—— **Joshua Bloch**

“初看本书书名时，我想，这是教人怎么入侵计算机系统的书吗？不太可能吧。嗯，那就肯定是一本编程小技巧的集锦。看了之后发现，没错，这就是一本编程秘籍，然而却是一本包罗万象的秘籍。第二版新增了两个大主题，并用数十个小技巧丰富了本书内容，其中有个小绝招是如何在不溢出的情况下求两数均值，我写二分查找算法时直接就把这条拿来用了。这真是本令算法爱好者开怀畅读的书啊！”

—— **Guy Steele**

在本书中，作者给我们带来了一大批极为诱人的知识，其中包括各种节省程序运行时间的技巧、算法与窍门。学习了这些技术，程序员就可写出优雅高效的软件，同时还能洞悉其中原理。这些技术极为实用，而且其问题本身又非常有趣，有时甚至像猜谜解谜一般，需要奇思妙想才行。简而言之，软件开发者看到这些改进程序效率的妙计之后，定然大喜。

本书较第1版增补了大量内容

- 新增了循环冗余校验（CRC）一章，其中讲解了常用的CRC-32校验码
- 新增了纠错码（ECC）一章，其中讲解了汉明码
- 详解了除数为常数的整数除法，增补了仅含移位操作和加法操作的算法
- 不计算商而直接求余数
- 扩充了与种群计数和前导0计数有关的知识
- 数组种群计数
- 执行压缩与扩展操作的新算法
- LRU算法
- 浮点数与整数互化
- 估算浮点数的平方根倒数
- 一系列离散函数图像
- 各章均配有习题与参考答案

PEARSON

ALWAYS LEARNING

机械工业出版社 China Machine Press

上架指导：计算机/算法

ISBN 978-7-111-45356-7

定价：89.00元

PEARSON

www.pearson.com

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn